

Eine Sicherheitsarchitektur für eine Informationssystem-Plattform

Diplomarbeit

vorgelegt von

César Perera Antón
Elektrotechnik
Technische Universität Hamburg-Harburg

Betreuer:

Prof. Dr. Florian Matthes
AB Software Engineering Betrieblicher Informationssysteme
TU München

Prof. Dr. Joachim W. Schmidt
AB Softwaresysteme
Technische Universität Hamburg-Harburg

Hamburg, den 19. Dezember 2003

Hiermit versichere ich, dass ich

1. diese Arbeit selbständig angefertigt,
2. alle wörtlich oder inhaltlich aus anderen Quellen entnommenen Stellen als solche kenntlich gemacht und
3. keine anderen als die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 19. Dezember 2003

César Perera Antón

Inhaltsverzeichnis

1	EINLEITUNG	1
1.1	ZIELE DER ARBEIT	2
1.2	GLIEDERUNG DER ARBEIT	3
1.3	DANKSAGUNG	3
2	GRUNDLAGEN	5
2.1	SICHERHEIT – BEGRIFFE, KONZEPTE UND METHODEN	5
2.1.1	<i>Allgemeines zum Thema Sicherheit.....</i>	<i>5</i>
2.1.2	<i>Sicherheitskonzepte.....</i>	<i>6</i>
2.1.2.1	Authentisierung	6
2.1.2.2	Autorisierung.....	7
2.1.2.3	Vertraulichkeit.....	7
2.1.2.4	Integrität.....	8
2.1.2.5	Verbindlichkeit.....	9
2.1.2.6	Weitere Sicherheitskonzepte	10
2.1.3	<i>Allgemeine Methodologie.....</i>	<i>11</i>
2.2	EIGENSCHAFTEN DER ZIELPLATTFORM	12
2.3	TECHNOLOGIEN	14
2.3.1	<i>Authentisierung.....</i>	<i>14</i>
2.3.1.1	Konzepte zur Authentisierung	14
2.3.1.2	Umsetzung webbasierter Authentisierung	17
2.3.2	<i>Kryptographie</i>	<i>22</i>
2.3.2.1	Überblick	22
2.3.2.2	Symmetrische Kryptographie	24
2.3.2.3	Asymmetrische Kryptographie	24
2.3.2.4	Hybride Chiffrierverfahren.....	25
2.3.3	<i>Digitale Zertifikate.....</i>	<i>25</i>
2.3.3.1	Konzepte	26
2.3.3.2	X.509 – Ein Standard für digitale Zertifikate	31
2.3.3.3	Zertifikatsspeicher in Java – serverseitige Betrachtung.....	32
2.3.3.4	Zertifikatsspeicher in Netscape – clientseitige Betrachtung.....	33
2.3.4	<i>Das SSL-Protokoll</i>	<i>34</i>
2.3.4.1	Der Protokollstapel – eine statische Betrachtung	35
2.3.4.2	Der SSL-Handshake – eine dynamische Betrachtung	36
2.3.4.3	SSL in Java – die Java Secure Socket Extension (JSSE)	42
2.3.5	<i>SafeGuard Biometrics – ein SmartCard-Produkt der Fa. Utimaco Safeware AG</i>	<i>44</i>
2.3.5.1	Produktauswahl	44
2.3.5.2	Produkteigenschaften	45
3	VERTRAULICHKEIT, INTEGRITÄT UND AUTHENTISIERUNG (V.I.A.).....	47
3.1	KONZEPTE ZUR UMSETZUNG BEIM INFOASSET BROKER.....	47
3.2	BESCHREIBUNG DES ENTWURFS.....	48
3.2.1	<i>Analyse des bestehenden Web Servers.....</i>	<i>48</i>
3.2.2	<i>Entwurf: Überlegungen und Kriterien.....</i>	<i>49</i>
3.2.3	<i>Entwurf: Die Gesamtarchitektur</i>	<i>52</i>
3.3	DETAILS DER IMPLEMENTIERUNG	56
3.3.1	<i>Dynamische Link-Erzeugung und JavaScript.....</i>	<i>56</i>
3.3.2	<i>Die Anmeldefunktionalität.....</i>	<i>59</i>
3.3.3	<i>Erkannte Lösungsdefizite</i>	<i>62</i>
3.4	ZUSAMMENFASSUNG	62

4	AUTORISIERUNG	63
4.1	KONZEPTION.....	63
4.1.1	Anforderungen.....	64
4.2	ENTWURF DER AUTORISIERUNGSRCHITEKTUR	66
4.2.1	Die Berechtigung.....	66
4.2.2	Die Definition von Berechtigungen.....	68
4.2.2.1	Programatische Definition der Berechtigungen	68
4.2.2.2	Deklarative Definition der Berechtigungen	70
4.2.3	Die Zuordnung von Berechtigungen zu Handlern	71
4.2.3.1	Fachlicher Entwurf	71
4.2.3.2	Technischer Entwurf.....	73
4.2.4	Die Auswertung von Berechtigungen.....	73
4.2.5	Die Gesamtarchitektur.....	74
4.3	DETAILS DER IMPLEMENTIERUNG	76
4.3.1	Die Initialisierung der Berechtigungen.....	76
4.4	ZUSAMMENFASSUNG.....	81
5	ZUSAMMENFASSUNG UND AUSBLICK	83
5.1	ZUSAMMENFASSUNG.....	83
5.2	AUSBLICK.....	83
5.2.1	Das Sicherheitskonzept Verbindlichkeit.....	83
5.2.2	Authentisierung: Verbesserungsvorschläge	84
5.2.3	Autorisierung: Verbesserungsvorschläge.....	85
	LITERATUR- UND QUELLENVERZEICHNIS.....	87

Abbildungsverzeichnis

ABB. 1:	INFOASSET BROKER - SCHICHTEN.....	12
ABB. 2:	AUTHENTISIERUNGSDREIECK	15
ABB. 3:	CHALLENGE-RESPONSE-PROTOKOLL (HASHFUNKTION MD5).....	18
ABB. 4:	AUTHENTISIERUNGSPROTOKOLL MIT DIGITALEM ZERTIFIKAT	20
ABB. 5:	KRYPTOGRAPHIE – KONZEPTUELLE GLIEDERUNG	23
ABB. 6:	ELEKTRONISCHES ARTEFAKT - DIGITALE ZERTIFIKAT	26
ABB. 7:	INTEGRITÄTSVERIFIZIERUNG VON ZERTIFIKATEN	27
ABB. 8:	ZERTIFIKATSSPEICHER IN NETSCAPE - SICHERHEITSEINRICHTUNGEN	34
ABB. 9:	SSL-PROTOKOLLSTAPEL	35
ABB. 10:	SSL-HANDSHAKE (CLIENT- UND SERVER-HELLO).....	36
ABB. 11:	SSL-HANDSHAKE (CLIENTSEITIGE SIGNATURPRÜFUNG).....	37
ABB. 12:	SSL-HANDSHAKE (HAUPTSITZUNGSSCHLÜSSEL).....	38
ABB. 13:	SSL-HANDSHAKE (SCHREIB- UND LESESCHLÜSSEL).....	38
ABB. 14:	SSL-HANDSHAKE (SERVER VERIFY 1/2).....	39
ABB. 15:	SSL-HANDSHAKE (SERVER VERIFY 2/2).....	39
ABB. 16:	SSL-HANDSHAKE (REQUEST CERTIFICATE)	40
ABB. 17:	SSL-HANDSHAKE (CLIENT CERTIFICATE 1/3)	40
ABB. 18:	SSL-HANDSHAKE (CLIENT CERTIFICATE 2/3)	41
ABB. 19:	SSL-HANDSHAKE (CLIENT CERTIFICATE 3/3)	41
ABB. 20:	JSSSE AUFBAU (HAUPTKLASSEN)	43
ABB. 21:	SMARTCARD-ARCHITEKTUR	45
ABB. 22:	ENTWURF DER SECURE WEB SERVER ARCHITEKTUR (SWS-ARCHITEKTUR).....	51
ABB. 23:	AUTORISIERUNGSRCHITEKTUR (BERECHTIGUNGEN).....	67
ABB. 24:	AUTORISIERUNGSRCHITEKTUR (SUBJEKTBASIERTE BERECHTIGUNGEN)	69
ABB. 25:	AUTORISIERUNGSRCHITEKTUR (ORGANISATIONSEINHEITEN).....	72
ABB. 26:	AUTORISIERUNGSRCHITEKTUR (GESAMTARCHITEKTUR)	75

1 Einleitung

Das Internet hat sich in den vergangenen 10 Jahren von einem Tummelplatz für Computerfreaks zu einem echten Massenmedium entwickelt. Es existiert mittlerweile kaum noch jemand, der nichts von dem Internet gehört hat, und trotz des Endes des Internet-Hypes und des Platzens der *dot-com*-Blase an den internationalen Finanzmärkten nimmt die Anzahl der Benutzer, die immer wieder neue Dienste im Internet für sich entdecken, kontinuierlich zu. Dies zeigt, dass das Internet seinen Platz unter den Massenmedien behauptet und ausbaut.

Das Internet revolutioniert unsere Art zu denken und zu handeln. Es tut dies, weil es in der Lage ist, sämtliche bisher bekannten Medien zu integrieren: Online-Zeitungen, Radio- und TV-Streaming, Videokonferenzen, Mailing, etc. Darüber hinaus ist es in der Lage, Abläufe des täglichen Lebens durch Online-Abläufe zu substituieren: Home-Banking, Online-Shopping, etc. Alle diese Anwendungen stellen für die Internetnutzer mittlerweile eine gewisse Selbstverständlichkeit dar.

In gleicher Weise wie die Popularität dieses relativ neuen Mediums zunimmt, nimmt auch die Qualität und die Vielfalt der im Internet angebotenen Dienste kontinuierlich zu. Während in der Anfangszeit des World-Wide-Web Internetpräsenzen durch statische Inhalte charakterisiert waren, erwartet der Benutzer heutzutage hochaktuelle, dynamische Inhalte. Dies spiegelt sich auch in der technologischen Entwicklung wider. Während eine statische Web-Site serverseitig mit einem einfachen Web-Server auskommt, erfordert eine interaktive, dynamische Web-Site zusätzlich mindestens noch einen serverseitigen Datenbank-Server mit einer geeigneten Skriptsprache (z.B. MySQL und PHP). Komplexe Anwendungen gehen noch einen Schritt weiter. Sie bauen oft auf mehrschichtigen, serverseitigen Architekturen auf. Internet-Portale sind typische Anwendungen, die von einer mehrschichtigen Architektur gebrauch machen. Sie aggregieren und ordnen verschiedene, korrelierte Dienste und bieten dem Benutzer auf diese Weise eine einheitliche Plattform an. Gleichzeitig stellen sie eine Fülle von Funktionen bereit, die administrative Aufgaben übernehmen und für den Benutzer transparent sind (Sessions-Management, Load-Balancing, etc.).

Funktionen, die weniger administrative Aufgaben und eher sicherheitsorientierte Aspekte berücksichtigen und die ebenfalls weitestgehend benutzertransparent sind, können ebenfalls integraler Bestandteil einer Portal-Software sein. Hierbei ist anzumerken, dass bei dem Thema Sicherheit jeder etwas anderes darunter versteht. Die nebulöse Frage, ob das Internet sicher ist oder nicht, lässt sich nicht auf simple Weise beantworten. Wenn man sich vergegenwärtigt, dass das Internet aus dem ARPA-Net hervorgegangen ist und bedenkt, dass das Ziel dieses militärischen Projekts war, ein ausfallsicheres Netzwerk zu schaffen, so kann mit Fug und Recht behauptet werden, dass das Internet sicher sei: Das Gesamtsystem Internet steht selbst dann noch zur Verfügung, wenn Teilsysteme (z.B. aufgrund eines atomaren Erstschlages) zusammengebrochen sind. Denkt man bei Sicherheit hingegen an die Sicherheit der eigenen Daten vor dem Einblick unbefugter Personen, so ist das Internet in der Hinsicht nicht ohne weiteres sicher.

Sicherheit spielt überall dort eine Rolle, wo wichtige, schützenswerte Güter existieren – sei es die nationale Sicherheit oder die Vertraulichkeit der eigenen Banktransaktion. In dieser Arbeit wird das Thema **Sicherheit** anhand einer spezifischen Portal-Software aufgegriffen: der Portalplattform **infoAsset Broker**.

1.1 Ziele der Arbeit

Die vorliegende Arbeit widmet sich dem Thema **Sicherheit** innerhalb eines sehr spezifischen Kontextes. Dieser Kontext wird durch die Rahmenbedingungen vorgegeben, die einerseits in typischen Internet-Protokollen verankert sind und die andererseits durch die festgelegten Server- bzw. Client-Applikationen bestimmt sind. Hierbei bildet der Application-Server – der **infoAsset Broker** – für den die Sicherheitsarchitektur gebaut wird, den Mittelpunkt der Betrachtungsweise, an dem sich Lösungsansätze und verwendete Technologien orientieren.

Als wesentlicher Ist-Zustand zu Beginn dieser Arbeit ist eine sehr geringe Penetration von Sicherheitsaspekten in der infoAsset Broker-Architektur zu konstatieren. Eine in allgemeiner Weise formulierte Zielsetzung dieser Arbeit ist es, diese Sicherheitsaspekte stärker in den Mittelpunkt zu rücken.

Geht man stärker in die Details, so gilt es zunächst zu analysieren, wo die Sicherheitslücken d.h. die potentiellen Gefahren des Systems liegen. Mittel zum Zweck sind innerhalb dieser Arbeit die Festlegung allgemein anerkannter Sicherheitskonzepte, anhand derer der Versuch unternommen wird, die Sicherheit des Systems anzuheben. Im Ergebnis soll man hierdurch die Umsetzung von vier wichtigen Sicherheitskonzepten erhalten: **Vertraulichkeit** und **Integrität** der übertragenen Daten zwischen Client und Server, eine differenzierte **Authentisierung** auf der Basis unterschiedlicher Authentisierungskonzepte und eine deklarative und erweiterbare **Autorisierung** festgelegter Prozesse.

Geht man ein wenig über die spezielle Umsetzung der genannten Sicherheitskonzepte hinaus, so bildet diese Arbeit ein Grundgerüst für die kontinuierliche Erweiterung der Sicherheitsanforderungen und Sicherheitsmaßnahmen beim infoAsset Broker. Aus diesem Grund geht diese Arbeit auf abstrakte Weise auf den Begriff Sicherheit ein: Beschäftigt man sich mit Sicherheit, so beschäftigt man sich mit Risiken oder Gefahren für ein definiertes System, die es vorausschauend abzuwehren gilt. Da für ein System stets neue Risiken oder Gefahren aufkeimen können, muss die Umsetzung von Sicherheit für ein System stets als Prozess verstanden werden. Aus diesem Grund beschränkt sich die Sichtweise innerhalb dieser Arbeit nicht ausschließlich auf die Umsetzung der vier genannten Sicherheitskonzepte.

Diese Sichtweise bestätigt sich durch ein Sicherheitskonzept, auf das innerhalb dieser Arbeit hingearbeitet worden ist, welches allerdings nicht umgesetzt worden ist: **Verbindlichkeit** (*Non-Repudiation*). Die Umsetzung und die Art der Umsetzung der vier Sicherheitskonzepte bilden eine nachhaltige Grundlage, um dieses Sicherheitskonzept erfolgreich umsetzen zu können.

1.2 Gliederung der Arbeit

Kap. 2 erfasst die Grundlagen, die zum Verständnis dieser Arbeit und der aufgezeigten Lösungen erforderlich sind. Diese umfassen zunächst die allgemeine Analyse des Sicherheitsbegriffs sowie der lösungsorientierten Sicherheitskonzepte. Sie umfassen zugleich die Analyse der Rahmenbedingungen, die durch den infoAsset Broker gegeben sind. Schließlich werden für die Integration der Sicherheitskonzepte Technologien verwendet, die ebenfalls innerhalb dieses Kapitels beschrieben werden.

Kap. 3 widmet sich der Integration der Sicherheitskonzepte Vertraulichkeit, Integrität und Authentisierung. Die Behandlung von drei Sicherheitskonzepten innerhalb eines einzigen Kapitels wird durch die Art der Umsetzung bedingt. Stark vereinfacht geht es hier um die Integration eines *Secure Web Servers* mit Client-Authentisierung. Als Schlüsseltechnologie ist hier die Anwendung des SSL-Protokolls zu nennen.

Kap. 4 widmet sich der Integration des Sicherheitskonzeptes Autorisierung. Innerhalb dieses Kapitels kommt es weniger darauf an, bestehende Technologien in die Architektur des infoAsset Broker zu integrieren. Vielmehr wird hier anhand des Autorisierungsbegriffs festgelegt, welche schützenswerte Ressourcen vorhanden sind und durch welche Mittel diese geschützt werden sollen.

Kap. 5 fasst schließlich die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf erkannte Möglichkeiten zur Erweiterung und Verbesserung der hier erarbeiteten Sicherheitsarchitektur.

1.3 Danksagung

Diese Arbeit ist im Rahmen einer Diplomarbeit im Arbeitsbereich Softwaresysteme der Technischen Universität Hamburg-Harburg entstanden. Ich danke allen Angehörigen dieses Arbeitsbereichs, die ich während meiner Arbeit kennen lernen durfte und die ein freundliches Arbeitsklima geschaffen haben.

Ganz besonderer Dank gebührt meinem Betreuer Dipl.-Inf. Patrick Hupe, der mich während meiner Arbeit in zahlreichen Diskussionen fachlich unterstützt hat. Darüber hinaus hat er durch seine fröhliche Wesensart stets zu meiner Motivation beigetragen.

2 Grundlagen

In diesem Kapitel werden zunächst alle Grundlagen beschrieben, die zum Verständnis der vorliegenden Arbeit erforderlich sind. Dabei wird im ersten Hauptabschnitt auf theoretische Grundlagen zum Begriff **Sicherheit** eingegangen. Der zweite Hauptabschnitt widmet sich den wesentlichen Eigenschaften des Zielsystems. Im dritten Hauptabschnitt wird schließlich auf erforderliche Technologien zur Umsetzung der Ziele eingegangen.

2.1 Sicherheit – Begriffe, Konzepte und Methoden

Dieser Abschnitt befasst sich konzeptionell mit dem Thema Sicherheit. Es wird zunächst auf einer abstrakten Ebene der Begriff **Sicherheit** definiert, und diese Definition anschließend für den IT-Bereich konkretisiert. Im Anschluss werden **Sicherheitskonzepte**, die für diese Arbeit eine besondere Bedeutung haben, ausführlich beschrieben, wobei auch auf Sicherheitskonzepte in deren Umfeld eingegangen wird. Schließlich wird in allgemeiner Weise auf eine mögliche Vorgehensweise bei der Konstruktion einer Sicherheitsarchitektur eingegangen.

2.1.1 Allgemeines zum Thema Sicherheit

Der Begriff **Sicherheit** ist weit verbreitet. Er wird innerhalb völlig verschiedener Themengebiete verwandt, so dass seine Semantik stark von dem jeweiligen Kontext abhängt. Um den Begriff in allgemeiner Weise verstehen zu können, ist es sinnvoll, sich zu vergegenwärtigen, dass der Begriff *Sicherheit* stets im Kontext wohldefinierter Gefahren, Bedrohungen oder Risiken zu sehen ist. Allgemein lässt sich sagen, dass es sich hierbei um unerwünschte Ereignisse handelt, deren Eintritt oder deren Eintrittsabfolge nicht deterministisch (also lediglich stochastisch) vorhergesagt werden können. Die Maßnahmen, die zur Verhütung des Eintritts eines oder mehrerer solcher Ereignisse dienen, sind in ihrer Gesamtheit dem Begriff *Sicherheit* untergeordnet. Oft bauen solche Maßnahmen aufeinander auf, greifen ineinander oder sind zueinander komplementär. Die Ungewissheit über die möglicherweise eintretenden Ereignisse, ihre Vielfalt und die Vielzahl der unterschiedlichen Maßnahmen, die zu ihrer Verhütung einsetzbar sind, machen das Thema *Sicherheit* in allgemeiner Weise sehr komplex.

Definition: Der Begriff **Sicherheit** umfasst die Gesamtheit der Maßnahmen, die zur Verhütung wohldefinierter, unerwünschter Ereignisse dienen, deren Eintritt oder deren Eintrittsabfolge nicht voraussagbar sind und die zu einer Schädigung von Ressourcen führen können.

Die zu betrachtenden Ereignisse, die man auch als Unwägbarkeiten bezeichnen kann, sind stets themenimmanent. Es ist deshalb sinnvoll, sich auf ein jeweiliges Themengebiet zu beschränken, um die jeweiligen Unwägbarkeiten detaillierter benennen zu können. In der vorliegenden Arbeit wird Sicherheit im Spannungsfeld der Informationstechnik betrachtet. Bei der Verhütung von Schädigung zu schützenden Ressourcen handelt es sich hierbei um die **Information** sowie die zu ihrer Verarbeitung nötigen **Infrastruktur**. Die Definition des Begriffs *Schädigung* nimmt in diesem Kontext besondere Ausmaße an, da man bereits von einer Schädigung der Ressource Information sprechen kann, wenn diese an dafür nicht vorgesehene Empfänger gelangt (Vertraulichkeit, Autorisierung), obwohl die betrachteten Ressourcen der Informationstechnik hierbei unverändert bleiben.

Definition: Bei der Vor-Schädigung zu schützenden Ressourcen im Kontext des Themengebiets Informationstechnik handelt es sich um die **Information** sowie die zu ihrer Verarbeitung nötigen **Infrastruktur**.

Um Sicherheitskonzepte und geeignete Schutzmechanismen erarbeiten zu können, ist es erforderlich, sich mit denkbaren Schädigungen im jeweiligen Themengebiet auseinanderzusetzen. Bei einer zu erstellenden Sicherheitsarchitektur werden anschließend erkannte Sicherheitskonzepte als Anforderungsgrundlage an das zu erstellende System benutzt. Die Umsetzung der Sicherheitskonzepte durch geeignete Schutzmechanismen kann zudem das Verständnis gewisser Sicherheitstechnologien erfordern. Auf beide Aspekte – Sicherheitskonzepte und erforderliche Technologien – wird in den folgenden Abschnitten näher eingegangen. Dabei wird der Schwerpunkt zielorientiert auf Sicherheit im Kontext webbasierter Plattformen für Unternehmensportale gesetzt.

2.1.2 Sicherheitskonzepte

Im vorangegangenen Abschnitt wurde der Sicherheitsbegriff in allgemeiner Weise und im Kontext der Informationstechnologie erläutert. Anschließend wurde kurz ein allgemeiner Prozess zur Erstellung einer Sicherheitsarchitektur dargestellt. Dieser Abschnitt befasst sich mit der Identifikation und der Beschreibung der Sicherheitskonzepte, von denen eine Teilmenge die Grundlage für das Anforderungsprofil der in dieser Arbeit dargestellten Sicherheitsarchitektur bildet.

Die hier dargestellten Sicherheitskonzepte sind im Kontext webbasierter Unternehmensportale zu sehen. Als Anwendungsszenario ist somit von einem Application Server, der seine Webdienste via TCP/IP.v4 der Netzgemeinde anbietet, auszugehen. Ferner ist davon auszugehen, dass es sich bei den Clients um Webbrowser handelt, die die angebotenen Dienste mit Hilfe typischer Anwendungsprotokolle (HTTP, FTP, usw.) über das weltweite Internet anfordern.

Die dargestellten Sicherheitskonzepte decken Risiken ab, die vordergründig und typisch für eine TCP/IP.v4-Client-Server-Architektur sind. Diese Auflistung erhebt deshalb keineswegs den Anspruch auf Vollständigkeit¹. Das Erkennen weiterer Risiken kann somit neue Sicherheitskonzepte zur Folge haben.

2.1.2.1 Authentisierung

Im weitesten Sinne bedeutet der Begriff **Authentisierung**, die Echtheit von etwas zu überprüfen. Im Internet ist jeder Rechner durch seine IP-Adresse charakterisiert. Da IP-Adressen über die Zeit aber unterschiedlichen Teilnehmern zugeordnet werden können, ist es für eine Webanwendung unmöglich festzustellen, welcher Teilnehmer sich hinter welcher IP-Adresse verbirgt. Der Teilnehmer kann sich zwar gegenüber der Webanwendung identifizieren, d.h. vorgeben, ein bestimmter Teilnehmer zu sein, die Webanwendung muss diese Identifikation allerdings verifizieren, d.h. ihren Wahrheitsgehalt überprüfen, um sicher zu sein, dass tatsächlich der vorgegebene Teilnehmer und kein anderer die Webanwendung benutzt. Dies kann beispielsweise sinnvoll sein, weil die Anwendung personalisierte Dienste anbietet. Überprüft ein Server die Identität eines Clients spricht man von **Client-Authentisierung**.

Oft ist auch das Pendant hierzu erforderlich. Teilnehmer greifen auf Webdienste zunächst durch die Eingabe eines Domännennamens zu. Ein solcher Domänenname muss in eine IP-Adresse aufgelöst werden, damit die Client-Anwendung weiß, zu

¹ Sie sind aus der Analyse verschiedener Quellen hervorgegangen (s. Literatur- und Quellenverzeichnis).

welchem Server sie ihren Request schicken soll. Zu diesem Zweck existieren sog. Domain-Name-Server (DNS), die diese Abbildung vornehmen. Es existiert allerdings keine Garantie, dass die gewünschte Domäne auch tatsächlich auf die richtige IP-Adresse abgebildet wird. Unter gewissen Umständen möchte der Teilnehmer allerdings sicher sein, dass er mit der von ihm gewünschten Webanwendung kommuniziert. In diesem Falle möchte der Teilnehmer die (von ihm vorgegebene) Identität der Webanwendung verifizieren, um sicher zu sein, dass tatsächlich die gewünschte Webanwendung und keine andere die angeforderten Dienste erfüllt. Dies kann beispielsweise sinnvoll sein, weil der Teilnehmer vertrauliche Daten an den Server schicken möchte. Überprüft ein Client die Identität eines Servers, spricht man von **Server-Authentisierung**.

Im engeren Sinne bedeutet *Authentisierung* somit, dass eine Identität verifiziert wird, d.h. es wird die Echtheit der vorgegebenen Identität überprüft.

Definition: Der Begriff **Authentisierung** umfasst sämtliche Maßnahmen, die dazu dienen, eine Identität durch geeignete Mittel zu verifizieren.

2.1.2.2 Autorisierung

Im weitesten Sinne bedeutet der Begriff **Autorisierung**, dass ein Recht zugebilligt worden ist, eine bestimmte Handlung durchzuführen. Woraus sich dieses Recht ableitet, kann dabei vielfältiger Natur sein.

Die im Internet von einem Server angebotenen Webdienste stehen grundsätzlich zunächst uneingeschränkt der gesamten Internetgemeinde zur Verfügung. Um einen solchen Dienst zu verwenden, muss ein potentieller Client seine Anforderung lediglich an den dienst anbietenden Server unter Verwendung des richtigen Protokolls und des richtigen Ports senden. Ob der Client zur Anforderung des gewünschten Dienstes berechtigt ist, wird von dem dienst anbietenden Server grundsätzlich in keiner Weise überprüft. In vielen Fällen ist es allerdings erforderlich, die zur Verfügung stehenden Ressourcen vor der Verwendung zu schützen. Ein solcher Verwendungsschutz wird in Form von Rechten dargestellt, die den einzelnen Ressourcen zugeordnet werden können und den Zugriff auf diese regeln. Im Allgemeinen ist es somit erforderlich, Rechte durch geeignete Maßnahmen zu definieren, die einzelnen Ressourcen auf Rechte abzubilden und die durch das zugehörige Recht beschriebenen Voraussetzungen beim Zugriff auf eine Ressource zu überprüfen. Die durch ein Recht beschriebenen Voraussetzungen können sich auf eine Vielzahl von Eigenschaften beziehen. Typische Eigenschaften sind die Identität des Benutzers bzw. seine Stellung innerhalb einer Organisation oder der Zustand der Ressource, auf die zugegriffen werden soll. Beruht die Autorisierung auf der Identität des Benutzers oder damit korrelierte Bereiche, so ist offensichtlich, dass eine Authentisierung des Benutzers hierfür eine Grundvoraussetzung ist.

Im engeren Sinne bedeutet *Autorisierung* somit, dass Zugriffsrechte definiert, diese den IT-Ressourcen zugeordnet und beim Zugriff auf letztere geprüft werden.

Definition: Der Begriff **Autorisierung** umfasst sämtliche Maßnahmen, die zur Definition, Zuordnung und Auswertung von Zugriffsrechten auf IT-Ressourcen dienen.

2.1.2.3 Vertraulichkeit

Im weitesten Sinne bedeutet der Begriff **Vertraulichkeit**, dass der Informationsgehalt eines Informationsträgers nur einer eingeschränkten (autorisierten) Gemeinde

zugänglich sein soll. Bei der Übertragung einer Information über das Internet wird immanent die aus den beiden Kommunikationspartnern bestehende Gemeinde verstanden. Wird beispielsweise von der *Vertraulichkeit* der auf einem Server liegenden Information gesprochen, so betrifft diese Fragestellung vordergründig die Autorisierung, da hier zunächst über das Zugriffsrecht auf die Information entschieden werden muss. Gemeint ist also diejenige *Vertraulichkeit*, die bei der Übertragung der Information zwischen den Kommunikationspartnern gewährleistet sein soll.

Die Natur des Internets bedingt zunächst, dass die übertragene Information ohne große Schwierigkeiten einer Fülle von Teilnehmern bekannt werden kann. Dies liegt darin begründet, dass die Vermittlung im Internet paketorientiert geschieht. Die übertragenen Pakete können von den Knoten, die diese Pakete weiterleiten, eingesehen werden, obwohl die enthaltene Information nur von dem Endempfänger einzusehen ist. Möchte man die *Vertraulichkeit* der Kommunikation gewährleisten, so ist die in den Paketen enthaltene Information vor dem Einblick der vermittelnden Knoten zu schützen. Dies kann beispielsweise sinnvoll sein, weil die übertragene Information ein Geheimnis darstellt, welches zur Authentisierung eines Teilnehmers dient (z.B. Passwort-Authentisierung).

Definition: Der Begriff ***Vertraulichkeit*** umfasst sämtliche Maßnahmen, die eine Information vor der Offenlegung gegenüber nicht autorisierten Personen schützen.

2.1.2.4 Integrität

Im weitesten Sinne bedeutet der Begriff ***Integrität***, dass die Aufrichtigkeit und die Zuverlässigkeit (i.d.R. einer Person) nicht angezweifelt wird.

Dieser Begriff ist hier in entsprechender Weise auf Informationsinhalte anzuwenden. Die Gewährleistung der Richtigkeit der Information steht hierbei im Vordergrund. Dies bedeutet insbesondere, dass die Information vor Manipulation durch Dritte aber auch vor zufälliger Veränderung geschützt wird, wie dies bei einer Informationsübertragung geschehen kann. Analog zum Begriff *Vertraulichkeit* ist beim Begriff *Integrität* wieder hauptsächlich die Kommunikation betroffen, da bei der Speicherung und dem potentiellen Zugriff wieder Autorisierungsfragen vordergründig zu betrachten sind.

Ebenso wie die Information während der Übertragung über das Internet Unbefugten bekannt werden kann (vgl. 2.1.2.3), kann die Information ggf. auf ihrem Weg manipuliert werden. Keine Schutzvorrichtung verhindert, dass die Information eines Pakets durch einen vermittelnden Knoten verändert wird. Möchte man die *Integrität* der Information sichern, so muss verhindert werden, dass Informationsinhalte der Pakete auf dem Weg vom Sender zum Empfänger verändert werden können bzw. dass eine Veränderung für den Empfänger als solche erkennbar ist, damit dieser die Information verwerfen und neu anfordern kann. Die Gewährleistung der *Integrität* der übertragenen Information kann beispielsweise deshalb wichtig sein, weil ihr Inhalt in eine wichtige Entscheidungsfindung einfließen kann, so dass eine falsche Information potentiell falsche Entscheidungen bedingen kann.

Definition: Der Begriff ***Integrität*** umfasst sämtliche Maßnahmen, die die Richtigkeit einer Information gewährleisten, insbesondere solche, die sie vor Manipulation oder zufälliger Veränderung schützen.

2.1.2.5 Verbindlichkeit

Im weitesten Sinne bedeutet der Begriff **Verbindlichkeit**, dass unveränderliche Bedingungen festgelegt worden sind, die die betreffenden Parteien zu einer Einhaltung und Erfüllung derselben verpflichten. Wie es dabei zu der Festlegung der Bedingungen gekommen ist, ist zunächst zweitrangig.

Der Gedanke, der mit diesem Konzept verfolgt wird, ist die Ermöglichung, einen digitalen Vertrag zwischen den Teilnehmern zu schließen bzw. die Ermöglichung von Rechtshandlungen auf digitalem Wege. Es dürfte offensichtlich sein, dass die bisher genannten Sicherheitskonzepte zur Erfüllung dieses Konzepts notwendig sind. So muss natürlich klar sein, wer einen Vertrag abschließt oder eine Rechtshandlung vollzieht (*Authentisierung*), es muss im Vorfeld des Abschlusses oder des Rechtshandlungsvollzugs geprüft werden, ob derjenige zu dieser Handlung berechtigt ist (*Autorisierung*) und die zu vereinbarenden Bedingungen dürfen während der Verhandlungen bzw. die Rechtshandlungen dürfen während ihrer Ausübung weder Unbefugten bekannt (*Vertraulichkeit*) noch von Saboteuren manipuliert (*Integrität*) werden. Darüber hinaus müssen vereinbarte Verträge oder vollzogene Rechtshandlungen juristisch ihrem nicht-digitalen Pendant gleichgestellt sein. Hierbei spielen zwei Dinge eine Rolle: Zunächst muss der Gesetzgeber solche digitalen Vereinbarungen bzw. Handlungen juristisch uneingeschränkt akzeptieren, d.h., dass es eine gesetzliche Regelung geben muss, damit solche Vereinbarungen oder Handlungen als Gegenstand einer gerichtlichen Auseinandersetzung zugelassen werden können.² Zum zweiten muss technisch nachvollziehbar sein, dass jemand einen solchen digitalen Vertrag eingegangen ist bzw. dass jemand eine Rechtshandlung auf digitalem Wege vollzogen hat, damit die Durchführung der entsprechenden Handlung nicht zurückgewiesen werden kann. Gegenstand dieser Arbeit sind natürlich nicht juristische, sondern technische Fragestellungen.

Im engeren Sinne befasst sich der Begriff **Verbindlichkeit** somit mit der Beantwortung der Frage, wie digitale Verträge bzw. vollzogene Rechtshandlungen festgehalten werden können, um später ggf. rechtlich darauf zurückgreifen zu können, und welche technischen Voraussetzungen hierfür notwendig sind, damit jemand, der einen Vertrag eingegangen ist oder eine Rechtshandlung vollzogen hat, diesen Sachverhalt nicht als unzutreffend zurückweisen kann. Dieses Konzept kann beispielsweise dann besonders wichtig sein, wenn Wirtschaftsgüter von besonders hohem Wert über das Internet gehandelt werden sollen (z.B. Immobilien) oder wenn politische Bürgerrechte auf neue Weise ausgeübt werden sollen (z.B. Volksentscheide bei wichtigen Gesetzen, vgl. Schweiz).

Definition: Der Begriff **Verbindlichkeit** umfasst sämtliche Maßnahmen, die zur Speicherung, Wiederverwertung und zum Nachweis der Durchführung einer elektronischen Rechtshandlung dienen. Ein Vertragsabschluss stellt hierbei eine besondere Rechtshandlung dar.

² In der Bundesrepublik Deutschland hat die Legislative mit dem Gesetz zur digitalen Signatur (SigG) vom 16.05.2001 hierfür die rechtlichen Voraussetzungen geschaffen.
(vgl. bundesrecht.juris.de/bundesrecht/sigg_2001/)

2.1.2.6 Weitere Sicherheitskonzepte

Über die zunächst ausführlich beschriebenen Sicherheitskonzepte hinaus existieren weitere nicht weniger wichtige Sicherheitskonzepte, deren Umsetzung nicht Gegenstand der vorliegenden Arbeit ist. Sie sollen hier allerdings benannt und kurz erklärt werden, um ein möglichst weitreichendes Bild der Sicherheitskonzepte darzustellen. Darüber hinaus ist es natürlich denkbar, dass bestehende Risiken noch nicht erkannt worden sind, und dass deshalb kein zugeordnetes Sicherheitskonzept bekannt ist. Aus diesem letzten Grund gibt die folgende Darstellung lediglich den aktuellen Stand der Dinge wieder.

Physische Sicherheit:

Dieses Sicherheitskonzept befasst sich mit den Maßnahmen, die ergriffen werden, um IT-Infrastruktur vor physischen Eingriffen oder unerlaubten physischen Zugängen zu schützen. Gemeint sind somit Zugangskontrollen zu IT-Anlagen, der Schutz der Energieversorgung der IT-Anlagen sowie evtl. Kühlvorrichtungen, der Schutz der IT-Anlagen vor Zerstörung, der Schutz der IT-Leitungen vor dem potentiellen Anzapfen, das Verhindern von Informationsgewinnung aus der elektromagnetischen Abstrahlung (Abschirmung) und dergleichen. Natürlich sind viele der hier als Beispiel genannten Maßnahmen nur insofern sinnvoll, wie der Wert der Information oder des IT-Dienstes dies rechtfertigt. Die Umsetzung dieser Maßnahmen erfordert zwar IT-Kenntnisse, sie sind allerdings nicht primär dem IT-Bereich zuzuordnen.

Netzwerksicherheit:

Dieses Sicherheitskonzept befasst sich mit Maßnahmen, die netzwerkimmanente Risiken abdecken, insbesondere, wenn es sich um offene Netze handelt, zu denen ein jeder Zugang hat, wie dies beim Internet der Fall ist. Hauptsächlich lassen sich die Risiken in zwei Kategorien gliedern: Einbruch und Sabotage.

Risiken aus der Kategorie Einbruch sind darauf ausgerichtet, unerlaubt an Information zu gelangen oder die IT-Infrastruktur zu eigenen, unerlaubten Zielen zu verwenden. Es entsteht hierbei kein offensichtlich sichtbarer Schaden, obwohl dieser natürlich vorhanden ist, wenn sensible Information in falsche Hände gerät oder wenn IT-Ressourcen zweckentfremdet werden. Derjenige, der einen solchen Übergriff begeht, hat i.d.R. ein Interesse daran, unerkannt und unbemerkt zu bleiben, so dass er seine Aktivitäten über einen längeren Zeitraum ausüben kann. Maßnahmen, die Risiken dieser Kategorie abwenden sollen, sind beispielsweise Firewalls, die offene Ports schützen oder dazu beitragen, Trojaner zu finden, die Planung der Netzwerkarchitektur unter Berücksichtigung der Einbruchsmöglichkeit (demilitarisierte Zonen, usw.), die Verwendung von Intrusion-Detection-Systemen, um Einbrüche sichtbar zu machen und evtl. zurückzuverfolgen, etc.

Risiken aus der Kategorie Sabotage sind darauf ausgerichtet, die originäre, funktionale Nutzung des Systems einzuschränken oder zu unterbinden. Sie sind i.d.R. darauf ausgerichtet, den vorhandenen IT-Ressourcen einen nachhaltigen Schaden zuzuführen. Beispiele für Risiken, die dieser Kategorie zuzuordnen sind, sind Viren, Würmer und Denial-of-Service Attacken (DoS bzw. dDoS).

Insgesamt kann man zusammenfassen, dass die Maßnahmen, die Netzwerksicherheit erzwingen sollen, nicht primär einen bestimmten Webdienst betreffen, sondern auf die Nutzung zusätzlicher IT-Infrastruktur sowie auf die sorgfältige Planung der Netzwerkarchitektur ausgerichtet sind.

Sicherheitsmanagement:

Dieses Sicherheitskonzept befasst sich mit sämtlichen Maßnahmen, die getroffen werden, um neue, zunächst unerkannte Risiken zu erkennen, zu benennen, zu analysieren und Strategien zu ihrer Verhütung zu entwickeln. Dieses Konzept führt zu einem Sicherheitsverständnis, bei dem Sicherheit nicht nur eine Fülle von Maßnahmen ist, sondern ein Prozess, bei dem die Sicherheitsanforderungen kontinuierlich angehoben werden, um das gewünschte System robuster gegen vorhandene Risiken zu machen.

Sicherheitsrichtlinien:

Dieses Sicherheitskonzept befasst sich mit der Definition von Vorgehensweisen zum Thema Sicherheit. Viele Sicherheitsmaßnahmen können leicht ausgehebelt werden, wenn die betreffenden Personen im Umgang mit diesen Maßnahmen anders vorgehen als dies zunächst vorgesehen ist. Sicherheitsrichtlinien versuchen dem entgegenzuwirken, indem sie den betreffenden Personen eine eindeutige Handlungsanweisung erteilen. Darüber hinaus kann für unterschiedliche Zwecke ein unterschiedliches Maß an Sicherheit erforderlich sein. Diese Unterschiede können durch unterschiedliche Sicherheitsrichtlinien ausgedrückt sein.

In diesem Zusammenhang ist hervorzuheben, dass durch die Festlegung von Sicherheitsrichtlinien die Benutzer eines technischen Systems, für das bestimmte Sicherheitsmaßnahmen implementiert werden, in dieses System mit einbezogen werden und die durch sie denkbar auszuführende Handlungen durch die Sicherheitsrichtlinien eingeschränkt werden. Im Allgemeinen lässt sich somit sagen, dass der Faktor Mensch im Umgang mit der Technik durch Sicherheitsrichtlinien Berücksichtigung findet. Ein einfaches Beispiel für die Einbeziehung des jeweiligen Benutzers in das System ist das Verbot zur Weitergabe von Passwörtern, PINs, SmartCards, etc. an Dritte, damit diese anstelle des Benutzers die jeweilige Handlung durchführen können.

2.1.3 Allgemeine Methodologie

Die vorangegangenen Abschnitte betrachten Sicherheit überwiegend statisch. Um ein System allerdings sicher zu machen, sollte ein Prozess durchlaufen werden und dieser ggf. iterativ, um bei jeder Iteration neue Sicherheitsrisiken zu berücksichtigen. Ein wichtiger Schritt eines solchen Prozesses ist die Risikoanalyse. In der Risikoanalyse wird zunächst untersucht, welche Gefahren die einwandfreie Funktionsweise des Systems in Frage stellen. Die erkannten Gefahren müssen hierbei bewertet werden, und es muss festgelegt werden, wie hoch der zu erreichende Sicherheitsgrad sein soll. Nachdem man in der Risikoanalyse eine erste grobe Einschätzung vorgenommen hat, kann man sich den Detailfragen widmen. Hierbei sind die zu schützenden Ressourcen, die Gefahren, die auf sie einwirken könnten, und die in Aussicht gestellten Sicherheitsmaßnahmen genauer zu benennen. Bei der Festlegung der Sicherheitsmaßnahmen ist natürlich der zuvor festgelegte Sicherheitsgrad zu berücksichtigen. Die festgelegten Sicherheitsmaßnahmen bilden gewissermaßen die Anforderungsspezifikation der zu erstellenden Sicherheitsarchitektur. Eine Erörterung der Stärken und Schwächen der eingesetzten Lösungskonzepte rundet hierbei die Risikoanalyse ab. Bei der gesamten Analyse ist auf ein ausgewogenes Maß zwischen Sicherheitsanforderungen, Benutzerfreundlichkeit und Kosten zu achten.

2.2 Eigenschaften der Zielplattform

In diesem Abschnitt wird ein kurzer Überblick über das bestehende Zielsystem gegeben, wobei sich der Schwerpunkt der Betrachtung auf diejenigen Aspekte konzentriert, die für diese Arbeit eine besondere Bedeutung haben.

Dem Zielsystem liegt eine internetbasierte Client-Server-Architektur zugrunde, bei der Clients durch gängige Browser verkörpert werden. Für den programmiertechnischen Teil dieser Arbeit wurde dabei Netscape 7.0 verwendet. Der Server besteht aus einem javabasierten *Application Server*, dem eine dreischichtige Architektur zugrunde liegt (Abb. 1) und der eine Portalplattform implementiert. Diese Portalplattform, die als **infoAsset Broker** (kurz Broker) bezeichnet wird, ist ein Softwareprodukt der Fa. infoAsset AG und stellt ein Framework für den Aufbau von internetbasierten Unternehmensportalen dar.

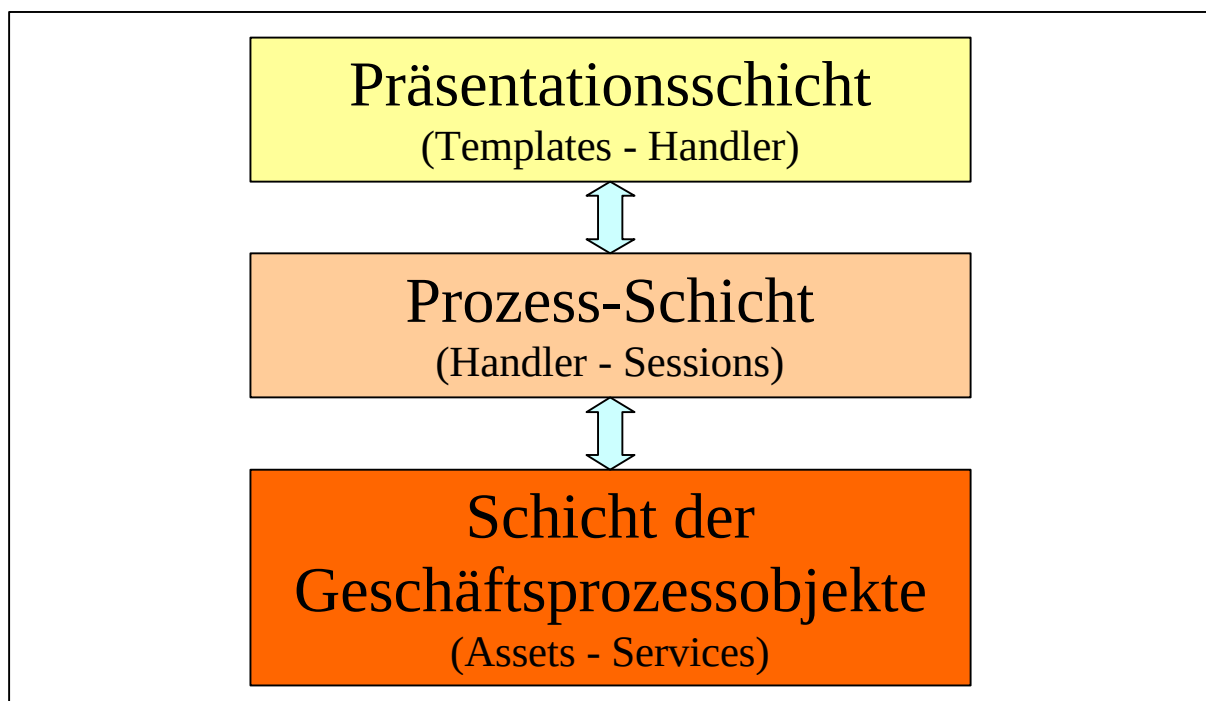


Abb. 1: infoAsset Broker - Schichten

Beim infoAsset Broker sind **Templates** im Wesentlichen statische HTML-Dokumente, die Platzhalter enthalten und beim Abruf durch den Client im jeweiligen Kontext dynamisch ersetzt werden. Hierbei existieren unterschiedliche Typen von Platzhaltern, die unterschiedliche Aufgaben erfüllen (Variablen, konditionale Platzhalter, Listen, etc.). Verknüpfungen innerhalb eines Dokuments werden durch einen mehrstufigen Mechanismus serverseitig auf **Handler** abgebildet. Die Handler enthalten die jeweilige Anwendungslogik. Sie rufen neue Handler auf, ersetzen Platzhalter in Templates mit aktuellen Daten in Abhängigkeit der jeweiligen **Session** und arbeiten auf den Geschäftsprozessobjekten. Diese Geschäftsprozessobjekte (*Business Objects*), die **Assets** genannt werden, die persistent gemacht werden können und auf die über festgelegte **Services** zugegriffen wird, werden durch vordefinierte Datenstrukturen festgelegt. Viele in Anwendungen oft benötigte Geschäftsprozessobjekte (z.B. Personen, Gruppen, Mitgliedschaften, etc.) sind beim infoAsset Broker bereits vordefiniert. Andere nicht vorhandene Typen können dem System nach einem bestimmten Schema hinzugefügt werden. Dies beschreibt grob die Funktionsweise des infoAsset Brokers.

Neben der dargestellten, groben Sichtweise sind weitere Sichten auf den Broker von maßgeblicher Bedeutung. So sind wesentliche Bestandteile dieser Arbeit (Kap. 3), vor dem Hintergrund zu verstehen, dass der eigentliche Web Server integraler Bestandteil des Brokers ist. Dies bedeutet, dass serverseitig ankommende Clientanfragen (*Requests*) direkt vom Java-Programmcode des Brokers verarbeitet und beantwortet werden.

Zwei weitere, für diese Arbeit wichtige Aspekte der Architektur der Portalplattform äußern sich clientseitig. Zum einen basiert das Layout der Webseiten auf *Frames*, zum anderen löst der Aufruf einer Verknüpfung (*Link*) im Regelfall die Ausführung von *JavaScript*-Programmcode im Browser aus. Diese beiden Aspekte scheinen belanglos und ohne wesentliche Auswirkungen auf die Sicherheitsarchitektur zu sein. Tatsächlich stellen sie eine nicht unwesentliche Hürde für die Integration der Sicherheitsarchitektur in die Portalplattform dar. Wie sich die Probleme hierbei konkret äußern und wie diese bewältigt werden, wird in Kap. 3 dargestellt.

Wie bereits erwähnt werden Verknüpfungen serverseitig auf Handler abgebildet. Die Zuordnung erfolgt deklarativ durch eine Datei namens *handler.txt*, die sich im jeweiligen Projektverzeichnis befindet. In dieser Datei werden die relativen Verknüpfungen, die i.d.R. vom Typ „Kategorie/Aktion“ bzw. „Kategorie/*“ sind, auf vollständige Java-Klassennamen (inkl. Package-Pfad) abgebildet. Hierbei muss die spezifizierte Klasse ein Handler sein, was technisch bedeutet, dass die benannte Klasse eine Spezialisierung der Klasse *GenericHandler* sein muss. Ein solcher Handler muss eine Methode namens *handleRequest* implementieren. Sie enthält den Programmcode mit der Anwendungslogik, der serverseitig ausgeführt wird, wenn eine Verknüpfung clientseitig aufgerufen wird.

Eine weitere Besonderheit existiert in diesem Kontext, da die Architektur der Portalplattform vorsieht, dass Projekte aufeinander aufbauen können. Als Grundprojekt ist hierbei stets, das Basisprojekt „Broker“ zu sehen, das bereits eine Fülle von Standardfunktionalität für Unternehmensportale bereitstellt. Die Architektur sieht vor, dass jedes Projekt eine eigene *handler.txt*-Datei im jeweiligen Projektverzeichnis enthält. Zuordnungen von Verknüpfungen zu Anwendungslogik werden somit auf jeder Projektebene definiert bzw. redefiniert. Hierbei haben Zuordnungen auf höherer Projektebene Priorität gegenüber den zugrunde liegenden Projekten. Die Zuordnungen im Basis-Projektverzeichnis „Broker“ greifen somit erst, wenn auf keiner der übergeordneten Projektebenen eine Zuordnung getroffen worden ist.

Dies sind die verschiedenen Sichten auf die Portalplattform, die für diese Arbeit von Bedeutung sind. Auf einzelne Details, die zusätzlich für die Lösung einzelner Probleme wichtig sind, wird dabei an gegebener Stelle eingegangen (Kap. 3 und Kap. 4)³.

³ für weitere Information zur Portalplattform siehe u.a. [iAB00]

2.3 Technologien

In diesem Abschnitt werden sämtliche technologischen Grundlagen der Sicherheitskonzepte dargestellt, die für diese Arbeit eine wichtige Rolle spielen.

2.3.1 Authentisierung

Dieser Abschnitt gibt zunächst einen Überblick, wie Authentisierung konzeptionell durchgeführt werden kann, um anschließend auf prinzipiell technische Umsetzungsmöglichkeiten einzugehen, die für diese Arbeit eine Rolle spielen.

2.3.1.1 Konzepte zur Authentisierung

Wie in Abschnitt 2.1.2.1 definiert, benötigt man Maßnahmen, um eine auf irgendeine Weise vorgegebene Identität zu verifizieren. Unter Identität ist hier hauptsächlich die Identität einer Person oder die Identität eines Hosts oder Webdienstes zu verstehen.

Die wichtigsten Authentisierungsmaßnahmen lassen sich für den spezifizierten Problembereich in drei Kategorien gliedern, wobei auch Mischformen häufig vorkommen. Jede Kategorie und jede Maßnahme für sich besitzt hierbei spezifische, konzeptionelle Vor- und Nachteile. Darüber hinaus kann eine Maßnahme, die aus konzeptioneller Sicht als besonders sicher zu bewerten ist, in ihrer technischen Umsetzung derartige Defizite aufweisen, dass sie aus technischer Sicht eine weitaus geringere Sicherheitsbewertung findet. Wie offenbar klar sein dürfte, setzt dies nicht die konzeptionellen Überlegungen in Frage, sondern ihre technische Umsetzung. Ein Beispiel hierfür wird weiter unten aufgeführt.

Für die exakte Bewertung einer spezifischen Authentisierungsmaßnahme sind zwei Messgrößen von Bedeutung: die Zurückweisungsrate und die Akzeptanzrate. Bei der Zurückweisungsrate ist zu messen, wie oft die Authentisierungsmaßnahme bei der Verifizierung der Identität zu einem negativen Ergebnis führt, obwohl es sich tatsächlich um die zu authentisierende Identität handelt, und somit das Ergebnis eigentlich positiv ausfallen müsste. Es wird somit die Wahrscheinlichkeit für die fälschliche Zurückweisung gemessen. Bei der Akzeptanzrate ist hingegen zu messen, wie oft die Authentisierungsmaßnahme bei der Verifizierung der Identität zu einem positiven Ergebnis führt, obwohl es sich nicht um die zu authentisierende Identität handelt, und somit das Ergebnis eigentlich negativ ausfallen müsste. Es wird somit die Wahrscheinlichkeit für die fälschliche Akzeptanz gemessen. Offenbar fällt eine Maßnahme in ihrer Bewertung besser aus je kleiner diese Größen sind. Solche Betrachtungen sind dann besonders wichtig, wenn der Authentisierungsgrad genau quantifiziert werden muss⁴ oder wenn bereits ein hoher Authentisierungsgrad erreicht ist, und die Einführung einer neuen Maßnahme zur Erhöhung der Sicherheit diskutiert wird. Leider sind solche quantitativen Betrachtungen allerdings hinfällig, wenn festgelegte Rahmenbedingungen (Sicherheitsrichtlinien) nicht eingehalten werden. Sind die bisherigen Sicherheitsmaßnahmen eines Systems als eher gering einzuschätzen, so ist es sinnvoller, den Sicherheitsgrad einzelner Authentisierungsmaßnahmen eher qualitativ zu bewerten.

Authentisierung beruht in vielen Fällen darauf, dass etwas vorgelegt wird, was die vorgegebene Identität bestätigt. Oft sind hierbei auch Merkmale vorhanden, die sich rein auf die Identifizierung beziehen und nicht zur eigentlichen Verifizierung dienen. Als Bezeichnung für die zur Authentisierung erforderliche Vorlage wird hier der Begriff **Credentials** gewählt.

⁴ z.B. um eine Spezifikation zu erfüllen

Definition: Der Begriff **Credentials** bezeichnet die Summe aller Elemente, die vorgelegt oder angewandt werden müssen, um eine erfolgreiche Authentisierung zu ermöglichen.

Wie eingangs erwähnt, sollen hier im Wesentlichen drei Authentisierungskategorien⁵ unterschieden werden: die Authentisierung kann grundsätzlich auf **Wissen**, **Besitz** oder **Biometrie** beruhen. Bei wissensbasierter Authentisierung wird davon ausgegangen, dass die sich authentisierende Identität über eine geheime Information verfügt, die sie authentisiert und die keiner anderen Identität bekannt sein darf. Ein typisches Beispiel hierfür ist das Passwort. Bei besitzbasierter Authentisierung ist die sich authentisierende Identität im Besitz eines Gegenstandes oder einer Information, deren Nutzung die Authentisierung herbeiführt. Ein typisches Beispiel für diese Art der Authentisierung sind digitale Zertifikate⁶ und ihr korrelierter, privater Schlüssel. Bei biometriebasierter Authentisierung ist die zu authentisierende Identität grundsätzlich eine Person, da Computer oder Dienste keine biometrischen Eigenschaften besitzen. Bei dieser Art der Authentisierung werden charakteristische Eigenschaften des Menschen für eine Authentisierung verwendet. Ein typisches Beispiel hierfür ist der Fingerabdruck. Diese drei Authentisierungskategorien lassen sich graphisch in einem Authentisierungsdreieck darstellen (Abb. 2).

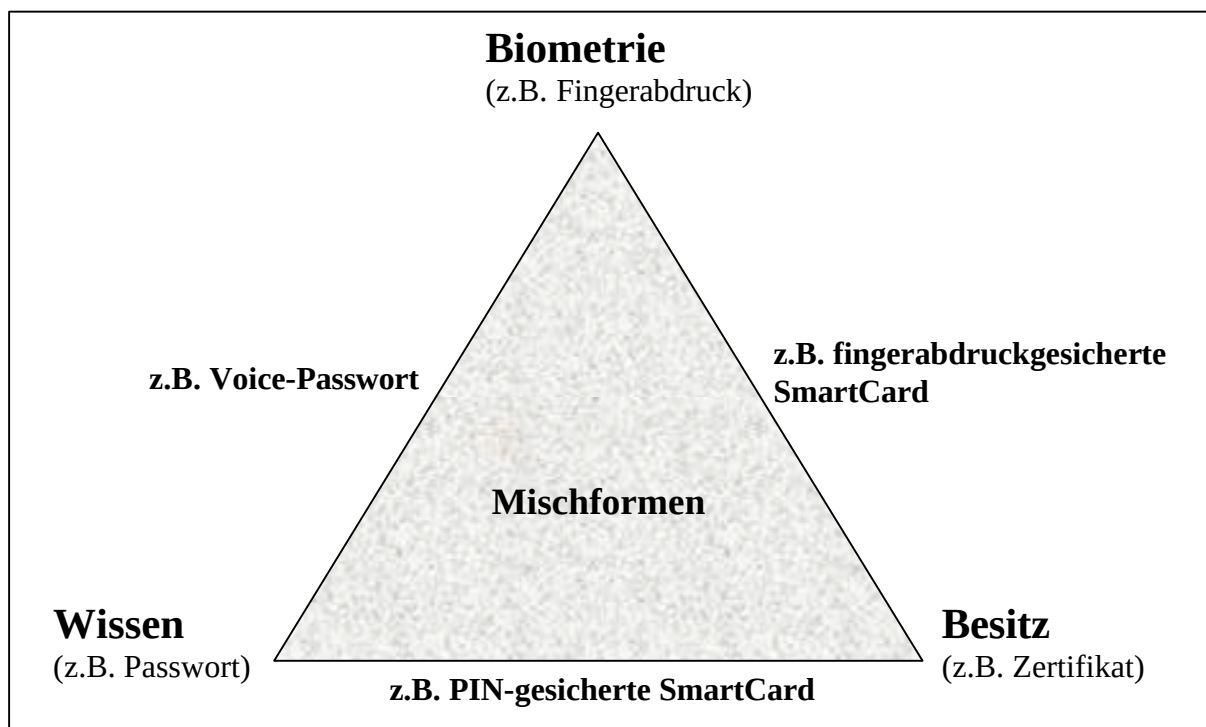


Abb. 2: Authentisierungsdreieck

Über die in Abb. 2 dargestellten Kategorien hinaus sind natürlich noch weitere denkbar. So werden beispielsweise von Brömme⁷ die Einbeziehung räumlicher und zeitlicher Aspekte vorgeschlagen. Solche Aspekte eignen sich gut, um die hier dargestellten Konzepte zu ergänzen. Sie spielen für die vorliegende Arbeit allerdings keine Rolle, weshalb auf sie auch nicht weiter eingegangen wird.

⁵ Auf weitere, denkbare Kategorien, die für diese Arbeit allerdings ohne Bedeutung sind, wird kurz eingegangen.

⁶ Auf digitale Zertifikate wird in einem eigenen Abschnitt näher eingegangen.

⁷ A. Brömme, ehem. Wissenschaftlicher Mitarbeiter an der Universität Hamburg, 2001

Wissensbasierte Authentisierung beruht auf der Tatsache, dass ein Geheimnis zwischen der Seite, die sich authentisieren möchte (Seite A), und der Seite, die die Authentisierung anfordert (Seite B), vorhanden ist. Das Geheimnis wird von A nach B übertragen, wodurch B feststellt, dass A Kenntnis des Geheimnisses hat. Daraus folgert B, dass A tatsächlich A sein muss, da ja nur A Kenntnis dieses Geheimnisses hat. Typische Beispiele für derartige Geheimnisse sind Passwörter, Persönliche-Identifikations-Nummern (PINs), Transaktions-Aktivierungs-Nummern (TANs), Antworten auf vorher festgelegte Fragen, verteilte Geheimnisse (*shared secrets*), etc. Einige der hier genannten Geheimnisse haben allerdings einen anderen Zweck als die Authentisierung, wie dies bei TANs und verteilten Geheimnissen der Fall ist. Dass diese Art der Authentisierung so weit verbreitet ist, hat einen simplen Grund: eine wissensbasierte Authentisierung ist mit wenig Aufwand umsetzbar. Dies ist wohl die größte Stärke dieses Konzepts. Auf der anderen Seite liegen die Schwächen auf der Hand: Geheimnisse können leicht weitergegeben oder kompromittiert werden. Ein hoher Standard bei den *Sicherheitsrichtlinien* (s. 2.1.2.6) und bei der *Vertraulichkeit* (s. 2.1.2.3) ist deshalb notwendig.

Besitzbasierte Authentisierung beruht auf der Tatsache, dass die Seite, die sich authentisieren möchte (Seite A) gegenüber einer anderen Seite (Seite B), im Besitz eines Gegenstandes oder Information ist, wobei die jeweilige Nutzung die Authentisierung hervorruft. Der wesentliche Unterschied zur wissensbasierten Authentisierung ist, dass ein evtl. bestehendes Geheimnis nicht zur Seite B übertragen wird. Es wird vielmehr das Anwendungsergebnis des Geheimnisses übertragen. In vielen Fällen ist ein solches Geheimnis auch nicht einem Anwender der Seite A bekannt, da es sich oft in einem Speicher befindet, der nicht ohne weiteres eingesehen oder kopiert werden kann. Typische Beispiele für derartige Gegenstände und Information sind der Zertifikatsspeicher eines Webbrowsers, der (eigene) digitale Zertifikate und die jeweils korrelierten privaten Schlüssel enthält, der Zertifikatsspeicher einer SmartCard, der die gleiche Funktion wie der des Webbrowsers erfüllt mit der zusätzlichen Sicherheit, dass die SmartCard dem System leicht entzogen und wieder hinzugefügt werden kann und dass der private Schlüssel auf einer SmartCard diese nicht verlässt, etc. Die wesentliche Stärke der besitzbasierten Authentisierung liegt in der Tatsache begründet, dass der zur Authentisierung erforderliche Gegenstand oder die erforderliche Information i.d.R. sehr schwer reproduzierbar ist bzw. sein sollte, wodurch man gegenüber der wissensbasierten Authentisierung qualitativ einen höheren Sicherheitsgrad erreicht. Wie bei der Erläuterung der digitalen Zertifikate noch gezeigt wird, ist das Verwenden digitaler Zertifikate mit einem nicht unerheblichen Aufwand verbunden. Darüber hinaus erfordert die Verwendung von Zertifikaten ein gewisses Grundwissen, so dass man sagen kann, dass ihre Verwendung weniger benutzerfreundlich ist. Bei der Verwendung von Zertifikaten auf SmartCards wird zudem noch weitere Hardware benötigt. All diese zuletzt genannten Eigenschaften haben dazu geführt, dass sich besitzbasierte Authentisierung⁸ clientseitig (noch) nicht durchgesetzt hat. Eine zunehmende Sensibilität für Sicherheitsaspekte könnte dies aber ändern.

Biometriebasierte Authentisierung beruht auf der Tatsache, dass ein Benutzer sich mit seinen charakteristischen, biometrischen Eigenschaften gegenüber seinem Kommunikationspartner authentisiert. Typische Beispiele hierfür sind der Fingerabdruck, die Stimmcharakteristik, die Gesichtsform, die Augen, die DNA, etc. Die besondere Stärke biometriebasierter Authentisierung ist die sehr hohe konzeptionelle Sicherheit dieses Ansatzes. So sind die genannten biometrischen

⁸ namentlich zertifikatsbasierte

Merkmale i.d.R. eineindeutig⁹ für die betreffende Identität. Die Schwächen liegen im oftmals hohen technischen Aufwand, der erforderlich ist, um die biometrischen Eigenschaften auf nahezu fälschungssichere Weise festzustellen. Darüber hinaus müssen die jeweils gemessenen biometrischen Eigenschaftsdaten mit Referenzdaten verglichen werden, die ihrerseits in einem sicheren Speicher gehalten werden müssen. Schließlich müssen die Messdaten unter Gewährleistung der Vertraulichkeit vom Messgerät zu diesem sicheren Speicher übertragen werden, um dort verglichen werden zu können. All dies spricht für eine lokale Anwendung, d.h. einem Verfahren bei dem Messdatengewinnung, Übertragung und Referenzdatenvergleich in einem Gerät stattfindet. Ein solches Gerät würde bei positivem Vergleich aktiviert werden, d.h. in einen Zustand versetzt werden, der einen erfolgreichen Vergleich der Mess- mit den Referenzdaten bestätigt, und durch die Aktivierung die Authentisierung hervorrufen.

2.3.1.2 Umsetzung webbasierter Authentisierung

Im vorangegangenen Abschnitt sind wesentliche Konzepte der Authentisierung vorgestellt worden. In diesem Abschnitt werden diese Konzepte aufgegriffen, und es wird diskutiert, wie die für diese Arbeit maßgeblichen technischen Umsetzungen im Detail aussehen. Hierbei werden im Einzelnen die technischen Prinzipien beschrieben, ohne jedoch auf Implementierungsdetails bei der Zielplattform einzugehen.

Wissensbasierte Authentisierung:

Eine der wohl bekanntesten wissensbasierten Authentisierungen ist diejenige, bei der die Credentials durch eine Benutzerkennung (*Userlogin*) und ein Passwort umgesetzt werden. Hierbei dient das Userlogin zur Identifikation des Benutzers und das Passwort, das ja ausschließlich diesem Benutzer bekannt sein soll, dient zum Nachweis der Identität, d.h. zur Authentisierung.

Problematisch bei der webbasierten Implementierung dieses Ansatzes ist die mangelnde Vertraulichkeit auf dem Übertragungskanal zwischen Client und Server. Wird das Passwort im Klartext übertragen, kann jeder, der die Kommunikation überwacht, sich das Passwort aneignen und sich fortan fälschlicherweise bei dem Zielsystem als der ursprüngliche Benutzer ausgeben. Wird das Passwort in einer chiffrierten, aber stets gleichen Weise übertragen¹⁰, kann die chiffrierte Information für einen sog. *Replay-Attack* verwendet werden. Hierbei kennt die unberechtigte Partei zwar das Passwort nicht, sie ist aber trotzdem in der Lage eine erfolgreiche Anmeldung durchzuführen, da sie dem Zielsystem genau die gleiche chiffrierte Information schicken kann wie der berechtigte Benutzer.

Eine einfache Lösung dieses speziellen Problems wird durch das *Challenge-Response-Protokoll* bereitgestellt (vgl. Abb. 3). Hierbei schickt der Server dem Client eine zufällig generierte Zahl (ein sog. *Nonce*¹¹). Der Client hängt an das Nonce das vom Benutzer eingegebene Passwort an, wendet auf das Ergebnis eine Hashfunktion an und überträgt den Hashwert zum Server. Der Server führt den analogen Vorgang serverseitig durch und vergleicht die beiden Hashwerte. Sind beide Hashwerte identisch, so kennt der Benutzer das erforderliche Passwort und ist somit authentisiert. Eine Kompromittierung des Passworts kann nicht stattfinden, da nicht das Passwort, sondern ein Hashwert übertragen wird. Die Rückgewinnung der

⁹ zumindest wenn man extreme Ausnahmen wie eineiige Zwillinge ausklammert

¹⁰ z.B. durch die Verwendung einer Hashfunktion

¹¹ i.e. Nonce = **n**umber **o**nce

Quellinformation aus einem Hashwert ist so gut wie unmöglich, da es sich bei Hashfunktionen um sog. Einwegfunktionen handelt. Auch ein Replay-Attack ist ausgeschlossen, da der Server für jeden Anmeldevorgang ein neues Nonce verwendet. Ein Angreifer müsste somit auf genau das Nonce warten, welches dem von ihm gelauschten Hashwert zugrunde liegt. Die Wahrscheinlichkeit eines Replay-Attacks ist umso näher an der Null, je größer der Zahlenraum ist, aus dem das Nonce stammt.

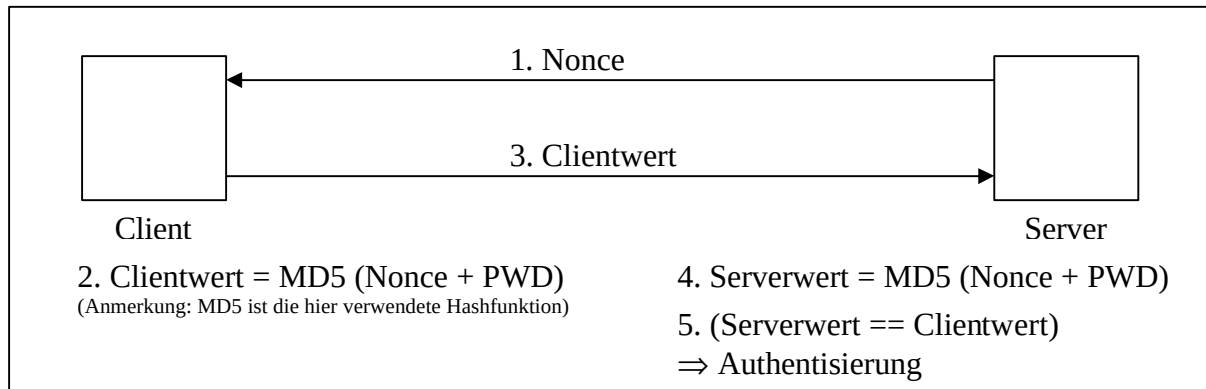


Abb. 3: Challenge-Response-Protokoll (Hashfunktion MD5)

Das Challenge-Response-Protokoll stellt eine Lösung für die Verifizierung der Kenntnis eines Passworts über einen nicht vertraulichen Kanal dar. In vielen Fällen ist es allerdings erforderlich, dass die gesamte oder große Teile der Kommunikation vertraulich bleiben. In solchen Fällen wird zunächst die Vertraulichkeit des Kommunikationskanals gesichert. Im Internet wird dafür häufig das SSL-Protokoll verwendet, auf das in 2.3.4 eingegangen wird. Über einen auf solche Weise gesicherten Kanal können auch Passwörter übertragen werden, da die Vertraulichkeit des Kanals die Kompromittierung des Passworts verhindert. In diesem Zusammenhang wird klar, dass die Qualität der Authentisierung auf der Qualität der Kanalvertraulichkeit beruht. Dies kann als Beleg für die eingangs gemachte Behauptung über die Wechselwirkung von Sicherheitsmaßnahmen (s. 2.1.1) gewertet werden.

Neben der Frage der vertraulichen Übertragung der Credentials ist auch die Frage nach der Gewinnung der Credentials zu stellen. In einer webbasierten Umgebung können hierfür zwei¹² unterschiedliche Ansätze verfolgt werden:

Bei dem ersten Ansatz kann ein individuell gestaltbares HTML-Formular verwendet werden. Das Userlogin und das Passwort werden dabei vom jeweiligen Benutzer in dafür vorgesehene Felder eingetragen, Formularvariablen zugeordnet und per POST- oder GET-Methode an den Server übertragen. Dieser Ansatz kann selbstverständlich auch im Zusammenhang mit einem Challenge-Response-Protokoll verfolgt werden. Da ein solches Abfrageformular im Prinzip auch nur eine HTML-Seite ist, kann ein potentieller Angreifer die Implementierung einer solchen Abfrage einsehen, analysieren und daraus Schlüsse ziehen. Er wäre somit in der Lage, einen kleinen Teilbereich der Sicherheitsarchitektur zu verstehen und möglicherweise mit weiteren Erkenntnissen auch in der Lage, Schwachstellen zu finden. Die Vorteile dieses Ansatzes sind die einfache Implementierung eines solchen Formulars und die individuelle gestalterische Freiheit.

¹² u.U. sind auch mehr denkbar

Bei dem zweiten Ansatz kann http-Headerinformation verwendet werden, um dem Browser des Clients mitzuteilen, dass eine Passwortauthentisierung erforderlich ist¹³. Hierbei versucht der Client zunächst auf eine Webseite zuzugreifen, für die eine Passwortauthentisierung erforderlich ist. Der Server beantwortet die Anforderung der Webseite (Request) mit einem Fehlercode (401) und mit einer Aufforderung im http-Header (WWW-Authenticate). Zusätzlich enthält der Header Information über das Schema der erwünschten Authentisierung und über den Bereich (*Realm*). Ein Browser, der einen solchen Header interpretieren kann, wird mit einer browsertypischen Eingabeaufforderung an den Benutzer reagieren. Der Benutzer kann in einem separaten Fenster seine Credentials eingeben und bestätigen. Nach der Bestätigung fordert der Browser genau dieselbe Seite wie zu Beginn an mit dem Unterschied, dass im http-Header die Credentials enthalten sind (WWW-Authentisation). Wie die Credentials codiert sind, hängt von dem Schema ab. Das gebräuchlichste Schema BASIC benutzt hierzu eine sog. Base64-Kodierung. Bricht der Benutzer hingegen die Eingabe der Credentials ab, so wird auf dem Browser die HTML-Information der Fehlerseite (401) wiedergegeben, die bis dahin zurückgehalten wurde. Ein erneuter Zugriff findet somit im Fall eines Abbruchs nicht statt.

Bei diesem zuletzt erklärten Ansatz ist eine clientseitige Informationsgewinnung bzgl. der Implementierungsdetails nicht möglich, da es sich hierbei um eine vom Browser bereitgestellte Funktion handelt, deren Implementierung nicht einsehbar ist. Die serverseitige Implementierung dieses Ansatzes ist dabei i.d.R. umständlicher, wie die Erläuterungen wiedergeben dürften. Auf Präsentationsebene erreicht man hierdurch ein einheitliches Authentisierungsbild, da das Eingabefenster browserspezifisch ist und keinerlei Gestaltung zulässt.

Besitzbasierte Authentisierung:

Die wohl bekannteste besitzbasierte Authentisierung im Kontext webbasierter Authentisierung beruht auf der Verwendung von digitalen Zertifikaten (s. 2.3.3). Die Verwendung digitaler Zertifikate, die nicht ausschließlich auf Authentisierung beschränkt ist, bildet einen wichtigen Bestandteil dieser Arbeit, weshalb Zertifikate weiter unten in eigenen Abschnitten erläutert werden. An dieser Stelle soll lediglich ihr Beitrag zur webbasierten Authentisierung verdeutlicht werden.

Ein digitales Zertifikat verknüpft auf untrennbare Weise eine Identität mit einem öffentlichen Schlüssel. Beide Informationen sind u.a. in einem Zertifikat eingetragen, wobei die Integrität des Zertifikats gesichert wird. Die Integrität des Zertifikats garantiert die Untrennbarkeit der beiden Informationen, wobei hier nicht darauf eingegangen wird, wie dies geschieht. Zu jedem öffentlichen Schlüssel existiert ein privater Schlüssel (s. 2.3.2.3), der ausschließlich dem Zertifikatsinhaber bekannt sein sollte. Die Credentials dieses Authentisierungsverfahrens bestehen aus dem digitalen Zertifikat und dem zugehörigen privaten Schlüssel, wobei die Vorlage des Zertifikats zur Identifizierung und die Anwendung des privaten Schlüssels zur Authentisierung dienen. Abb. 4 verdeutlicht das Authentisierungsprotokoll in generischer Weise.

¹³ [Wong00]

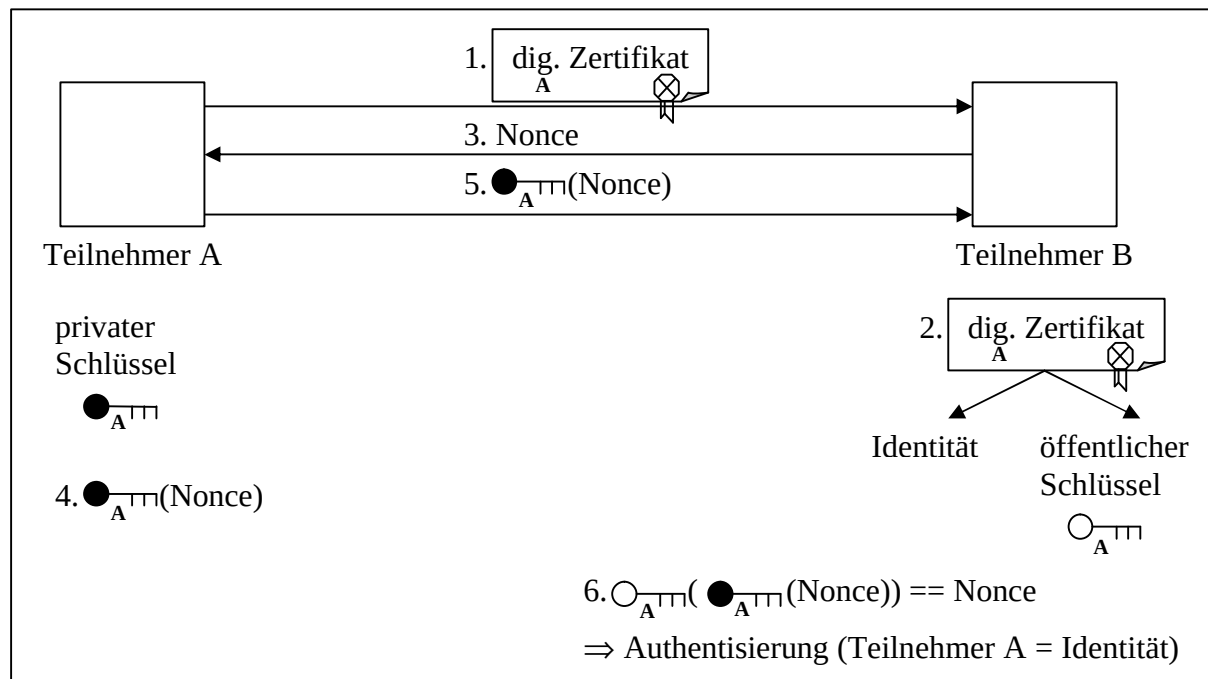


Abb. 4: Authentisierungsprotokoll mit digitalem Zertifikat

Zunächst überträgt derjenige Teilnehmer A, der sich authentisieren möchte, sein Zertifikat an denjenigen Teilnehmer B, der die Authentisierung anfordert. Teilnehmer B extrahiert die in dem Zertifikat eingetragene Identität und ihren öffentlichen Schlüssel. Teilnehmer B schickt Teilnehmer A ein Nonce, das von letzterem mit seinem privaten Schlüssel digital signiert werden muss. Da nur der Zertifikatsinhaber den privaten Schlüssel kennt, ist dieser als einziger in der Lage, diese Signatur zu leisten. Nun schickt Teilnehmer A Teilnehmer B das signierte Nonce. Teilnehmer B wendet den aus dem Zertifikat extrahierten öffentlichen Schlüssel auf das signierte Nonce an. Für den Fall, dass das ursprünglich an Teilnehmer A gesendete Nonce tatsächlich mit dem zum Zertifikat zugehörigen privaten Schlüssel signiert wurde, ergibt die Anwendung des öffentlichen Schlüssels auf das signierte Nonce wieder das ursprüngliche Nonce. Da nur der Zertifikatsinhaber diesen privaten Schlüssel kennt und somit als einziger in der Lage gewesen sein kann, die erforderliche Signatur zu leisten, schließt Teilnehmer B (im erfolgreichen Fall), dass Teilnehmer A und der Zertifikatsinhaber miteinander identisch sind, wodurch die Authentisierung gegeben ist. Es wird ergänzend darauf hingewiesen, dass keinerlei geheime Information über den Kanal übertragen wird, und damit kann die Gefahr der Kompromittierung dieser geheimen Information (hier: privater Schlüssel) als deutlich geringer als bei der wissensbasierten Authentisierung eingeschätzt werden.

Bewertung: Die Qualität der Authentisierung wird in unterschiedlicher Weise durch die Qualität des Zertifikats geprägt. Diejenige Kommunikationsseite, die die jeweilige Identität überprüft (hier: Teilnehmer B), muss anhand des vorgelegten digitalen Zertifikats entscheiden, ob dieses Zertifikat eine hinreichende Authentisierung ermöglicht. Diese Entscheidung ist anhand anderer in das Zertifikat eingetragener Parameter zu treffen, wobei sie i.d.R. beruhend auf vorgenommenen Einstellungen in automatisierter Weise erfolgt. Auf diesbezügliche Details wird weiter unten eingegangen.

Die Qualität der Authentisierung kann sich zusätzlich durch den Speicherort der Credentials sowie durch seine lokale Zugriffssicherung unterscheiden. Als Speicherort kann beispielsweise eine verschlüsselte Datei auf einer lokalen

Festplatte, auf einer Netzwerkplatte, in einer Datenbank, auf einer Diskette, usw. dienen. Problematisch bei diesem Ansatz ist die Tatsache, dass Dateien leicht zu kopieren sind und dass die Datei, die die Credentials enthält, somit leicht dupliziert und missbräuchlich verwendet werden kann. Abhilfe schaffen hier Datenträger, die keine Duplizierung zulassen, weil die geheime Information (hier: privater Schlüssel) auf den Datenträgern entsteht, ausschließlich in ihnen verwendet wird und diese niemals verlässt. Ein typisches Beispiel für einen solchen Datenträger ist die *SmartCard*, die tatsächlich weit mehr als ein simpler Informationsspeicher ist. Eine SmartCard ist ein Mikrocontroller im Scheckkartenformat, der für spezifische Aufgaben ausgelegt wird. Bei der besitzbasierten Authentisierung soll die SmartCard die jeweiligen Credentials speichern, wobei die geheime Information die SmartCard niemals verlassen darf. Auf diese Weise ist der Besitz der SmartCard mit dem Besitz der Credentials identisch, was bei der besitzbasierten Authentisierung beabsichtigt wird. Diese Sichtweise ist bei Credentials, die in Dateien gehalten werden, aufgrund der Kopierfähigkeit der Dateien nicht so eindeutig.

Besitzbasierte Authentisierung – Lokaler Zugriffsschutz:

Wie zu Beginn des letzten Absatzes angedeutet, ist nicht nur die Verkörperung der Credentials entscheidend, sondern auch ihr lokaler Zugriffsschutz. Damit ist der Mechanismus gemeint, der vorhandene Credentials verwendbar macht. So werden die Credentials in Dateien häufig wissensbasiert geschützt, d.h. i.d.R. durch die Eingabe eines Passworts. Dieses Passwort wird allerdings nicht zum Kommunikationspartner übertragen, um eine Authentisierung zu bewirken, sondern es wird verwendet, um die Credentials, die zur Authentisierung verwendet werden, lokal freizuschalten. Dieses Prinzip wird von gängigen Browsern verwendet, wenn es darum geht, eigene Zertifikate und private Schlüssel zu verwenden. Es soll sicherstellen, dass der lokale Speicherort der Credentials tatsächlich seinem Benutzer zuzuordnen ist. Auch bei der Verwendung von SmartCards findet man ähnliche Prinzipien wieder. So kann der Zugriff auf die in einer SmartCard gehaltenen Credentials ebenfalls wissensbasiert (z.B. PIN-gesicherte SmartCard) oder sogar biometriebasiert (z.B. fingerabdruckgesicherte SmartCard) geschützt werden. Wie deutlich wird, erweitert der lokale Zugriffsschutz das reine Besitzkonzept des digitalen Zertifikats um andere Authentisierungskonzepte wie Wissen oder Biometrie.

Oft wird in diesem Kontext darüber diskutiert, welches dieser beiden lokalen Zugriffsschutzkonzepte denn das Bessere sei. Für beide Konzepte können jeweils gute Gründe vorgebracht werden. Zugleich können auch die Nachteile des nicht favorisierten Konzepts benannt werden. Diese Diskussion soll kurz anhand des Beispiels einer PIN-gesicherten und einer fingerabdruckgesicherten SmartCard skizziert werden. Die Befürworter einer PIN-gesicherten Lösung weisen oft darauf hin, dass der entsprechende Fingerabdruck des jeweiligen SmartCard-Inhabers leicht der Karte entnommen werden könne, da dieser die Karte ja berühren müsse, und dass diese Fingerabdruckinformation mit entsprechenden technischen Mitteln¹⁴ für eine Replay-Attacke verwendet werden könne. Die Ermittlung der jeweiligen PIN sei im Vergleich zum Fingerabdruck, schwieriger zu bewerkstelligen. Dem kann man entgegenhalten, dass sich zwar das technische Fingerabdruckerfassungssystem möglicherweise täuschen lasse, dies aber lediglich schlussfolgern lasse, dass solche Systeme noch nicht technisch ausgereift seien. Auf konzeptioneller Ebene hingegen sei das Fingerabdruckkonzept besser als das PIN-Konzept, da der erforderliche

¹⁴ hierbei kommen beispielsweise Silikonfingerreproduktionen zum Einsatz

Zugriffsschutz nicht auf einfache Weise freiwillig an einen Dritten¹⁵ weitergegeben werden könne. Darüber hinaus könne der Finger nicht „vergessen“ oder „verlegt“ werden, wie es im Falle einer PIN durchaus möglich sei, was erhebliche Verwaltungskosten einspare.

Zu erwähnen bleibt in diesem Kontext, dass zertifikatsbasierte Authentisierung von dem wohl verbreitetsten Sicherheitsprotokoll im Internet – dem SSL-Protokoll – unterstützt wird, und zwar sowohl client- als auch serverseitig. Der Einsatz dieses Standards bildet ebenfalls einen weiteren Eckpfeiler dieser Arbeit. Leider berücksichtigt das SSL-Protokoll hierbei nicht den Speicherort und den jeweiligen lokalen Zugriffsschutz der Credentials, so dass Lösungen hierfür auf darüberliegenden Anwendungsebenen bereitgestellt werden müssen.

Biometriebasierte Authentisierung:

Eine reine biometriebasierte Authentisierung im Kontext webbasierter Authentisierung ist zwar technisch denkbar, allerdings kommt sie praktisch nicht bzw. noch nicht vor. In dieser Arbeit tritt eine biometriebasierte Authentisierung nur als lokaler Zugriffsschutz auf, und zwar in der Weise, dass digitale Zertifikate und zugehörige private Schlüssel auf einer verwendeten SmartCard mit Hilfe von Fingerabdruckdaten gesichert werden. Das externe SmartCard-Lesegerät enthält zu diesem Zweck einen integrierten Fingerabdrucksensor. Bei einer Verwendung der SmartCard-Credentials wird der Benutzer aufgefordert, sich gewissermaßen gegenüber der SmartCard zu authentisieren, damit diese die Webauthentisierung einleiten kann. Die Referenzdaten der Fingerabdrücke befinden sich auf der SmartCard gespeichert und werden auch auf dieser mit den jeweiligen aktuell ausgelesenen Daten verglichen. Damit existiert zwischen dem Benutzer, dessen Identität nachgewiesen werden soll, und den Credentials, mit denen dies webbasiert bewerkstelligt werden soll, folgende Zuordnung:

Benutzer ↔ Fingerabdruck ↔ SmartCard ↔ Credentials (Zertifikat & priv. Schl.)

Die Zuordnungskette verdeutlicht, dass nur derjenige Benutzer die Credentials auf der Karte verwenden kann, der auch den richtigen Fingerabdruck nachweisen kann und dies kann lediglich derjenige, für den die Karte ausgestellt wurde. Der Besitz der Karte alleine ermöglicht somit noch keine erfolgreiche Authentisierung. Darüber hinaus ist eine Vervielfältigung der Credentials so gut wie unmöglich, da gewisse Information (hier: priv. Schlüssel) die Karte niemals verlässt. Auf diese Weise wird eine sehr gute Bindung eines Benutzers an seine Credentials erreicht.

2.3.2 Kryptographie

Die Kryptographie ist eine Schlüsseltechnologie, wenn es darum geht, die eingangs beschriebenen Sicherheitskonzepte umzusetzen. Das Thema Kryptographie ist sehr umfangreich und eine ausführliche Behandlung würde mit Sicherheit den Rahmen dieser Arbeit sprengen¹⁶. Da einige Aspekte allerdings besonders wichtig für das Verständnis dieser Arbeit sind, sollen sie kurz dargestellt werden.

2.3.2.1 Überblick

Kryptographie ist eine Wissenschaft, die die Verschleierung des Informationsgehaltes einer Botschaft gegenüber Dritten sowie die unberechtigte Rückgewinnung dieses Informationsgehaltes durch Dritte zum Ziel hat. Berechtigt ist

¹⁵ Umgehung bestehender Sicherheitsrichtlinien

¹⁶ für eine ausführliche Darstellung siehe u.a. [Zim97]

hier derjenige, für den die Botschaft oder Information bestimmt ist. Es wird ausdrücklich darauf hingewiesen, dass nicht die Existenz einer Information oder Botschaft verschleiert wird, sondern ihre Semantik. Diese Eigenschaft grenzt die Kryptographie gegenüber einer verwandten Wissenschaft – der **Steganographie** – ab, bei der bereits die Existenz der Information vor unberechtigten Dritten verschleiert wird. Beide Wissenschaften sind der **Kryptologie** untergeordnet, die die Verschleierung von Information generell zum Ziel hat.

Bei kryptographischen Verfahren wird Information in allgemein verständlicher Weise (Klartext) auf eine verschlüsselte Informationsrepräsentation abgebildet (Chiffre) und vom berechtigten Empfänger wieder zurückverwandelt, ohne dass die Semantik der Information verloren geht. Zu diesem Zweck werden parametrisierte Funktionen eingesetzt, bei denen der Funktionsparameter einen Schlüssel für die jeweilige Transformation bildet (s. Funktionsabbildung in 2.3.2.2). Die hierfür erforderlichen Abbildungen gliedern sich prinzipiell in zwei Kategorien: Bei der **symmetrischen Kryptographie** sind der Schlüssel für den Chiffrierschritt und der Schlüssel für den Dechiffrierschritt identisch, so dass dieser sowohl Sender als auch Empfänger bekannt sein muss. Bei der **asymmetrischen Kryptographie** sind die zuvor genannten Schlüssel grundsätzlich verschieden, so dass es einen Chiffrierschlüssel und einen komplementären Dechiffrierschlüssel gibt.

Ein dritter Bereich der Kryptographie – **die Kryptoanalyse** – befasst sich mit der Dechiffrierung ohne Kenntnis des erforderlichen Schlüssels und der Entwicklung von Verfahren, die dem dienlich sind. Die Kryptoanalyse verfolgt hierbei zwei Absichten: Die erste ist die Analyse der Sicherheit der kryptographischen Verfahren, die man selbst einsetzt. Die zweite ist schlichtweg das Aufdecken fremder Information, um sie sich i.d.R. (in unberechtigter Weise) zunutze zu machen.

Abb. 5 zeigt ein Organigramm mit den beschriebenen Begriffen. Die Themen, die für diese Arbeit von besonderer Bedeutung sind, wurden in Fettschrift hervorgehoben.

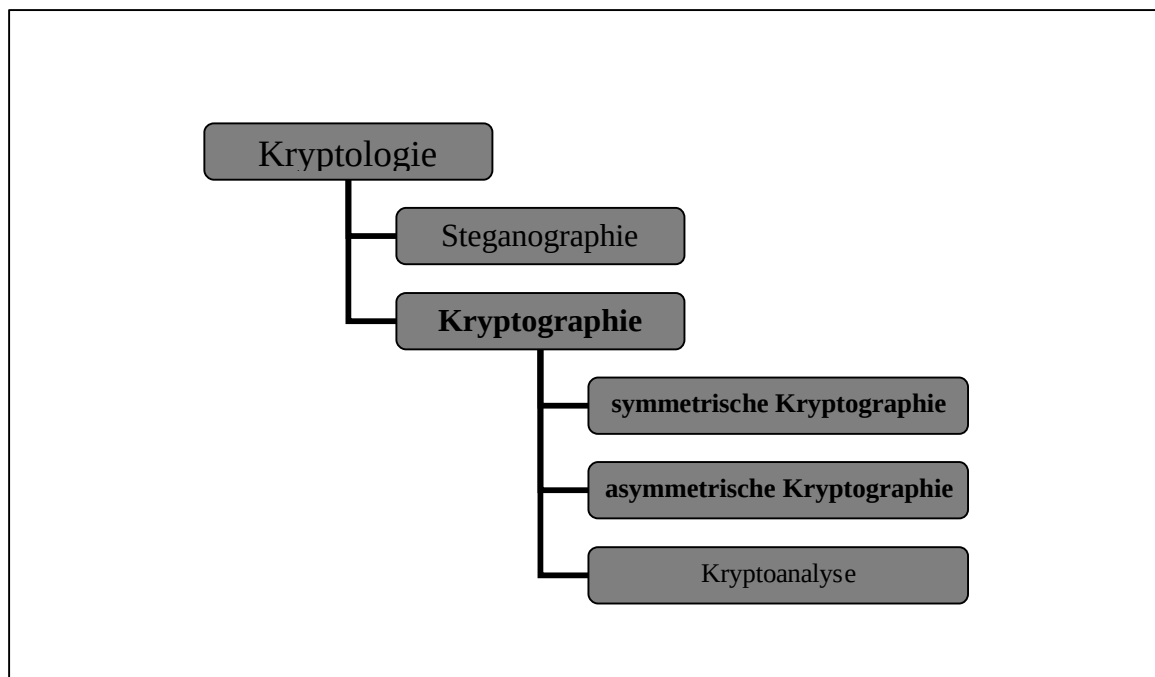


Abb. 5: Kryptographie – konzeptuelle Gliederung

2.3.2.2 Symmetrische Kryptographie

Bei einer symmetrischen Chiffrierung ist der Funktionsparameter (Schlüssel) der Verschlüsselungsfunktion für Chiffrier- und Dechiffrierschritt identisch. Die Chiffrierfunktion F bildet mit Hilfe des Schlüssels K einen Klartext x auf einen chiffrierten Text y ab. Dies bezeichnet man als Chiffrierschritt: $F : (x, K) \rightarrow y$. Die inverse Funktion F^{-1} bildet mit Hilfe des gleichen Schlüssels K den chiffrierten Text y auf den Klartext x ab. Dies bezeichnet man als Dechiffrierschritt: $F^{-1} : (y, K) \rightarrow x$. Eine gute Verschlüsselungsfunktion F zeichnet sich dadurch aus, dass eine Ermittlung des Klartextes x aus dem chiffrierten Text y ohne Kenntnis des Schlüssels K nicht ohne erheblichen Aufwand möglich ist (im Idealfall ist die Ermittlung unmöglich). Darüber hinaus sollten die Funktionswerte der Funktion F sowie ihrer Inversen F^{-1} mit Kenntnis des Schlüssels K auf einfache Weise zu berechnen sein, damit das Chiffrierverfahren performant ist.

Das Chiffrierverfahren wird durch die Funktion F charakterisiert, wobei diese i.d.R. bekannt ist. Lediglich der Schlüssel K wird unter den Kommunikationsteilnehmern geheim gehalten. Dies ist der Grund warum man im Kontext symmetrischer Kryptographie von **geheimen Schlüsseln** (*secret key*) spricht.

Symmetrische Chiffrierverfahren werden i.a. zum vertraulichen Informationsaustausch während einer Sitzung verwendet. Dabei wird der Klartext i.d.R. in festen Blocklängen chiffriert (z.B. 64 Bit). Die Länge des Schlüssels wird dabei häufig als Maß für die Sicherheit des Verschlüsselungsverfahrens angesehen, da bei einem exhaustiven Angriff auf eine verschlüsselte Information sämtliche denkbaren Schlüssel ausprobiert werden. Die Anzahl der Schlüssel wächst dabei exponentiell mit der Anzahl der Bits, die zur Kodierung eines Schlüssels erforderlich sind. Zurzeit wird eine Schlüssellänge von 128 Bit als hinreichend sicher angesehen. Es existiert eine Fülle symmetrischer Chiffrierverfahren, von denen hier nur einige benannt werden: DES, AES, RC4, usw.

Symmetrische Chiffrierverfahren sind i.d.R. gegenüber asymmetrischen Verfahren sehr performant. Ihre Schwäche liegt in der Notwendigkeit, einen geheimen Schlüssel über einen bereits sicheren Kanal vereinbaren zu müssen. Darüber hinaus sind Schlüssel der genannten Länge tatsächlich nur für die Dauer einer Sitzung hinreichend sicher, über einen längeren Zeitraum könnten die Schlüssel leicht kompromittiert werden. Schließlich wird für je zwei Teilnehmer eines Kommunikationsnetzwerks ein geheimer Schlüssel benötigt. Dies bedeutet eine quadratische Zunahme der Schlüssel mit der Anzahl der Teilnehmer, was bei großen Netzwerken wie dem Internet zu Problemen führen würde.

2.3.2.3 Asymmetrische Kryptographie

Bei einer asymmetrischen Chiffrierung sind der Schlüssel für den Chiffrierschritt und der Schlüssel für den Dechiffrierschritt grundsätzlich verschieden. Sie sind bzgl. des jeweiligen Verfahrens zueinander komplementär, d.h. was man mit dem einen Schlüssel chiffriert, wird mit dem komplementären Schlüssel dechiffriert. Mit welchem der beiden Schlüssel eine Chiffrierung stattfindet, hängt von der jeweiligen Anwendung ab.

Bei asymmetrischen Chiffrierverfahren hat jeder Teilnehmer eines Kommunikationsnetzwerks zwei Schlüssel: einen **öffentlichen Schlüssel** (*public key*) K_{pub} und einen komplementären **privaten Schlüssel** (*private key*) K_{priv} . Der

öffentliche Schlüssel eines Teilnehmers ist allen anderen Teilnehmern des Netzwerks zugänglich, während der private Schlüssel lediglich dem Teilnehmer selbst bekannt ist. Findet eine Chiffrierung mit dem öffentlichen Schlüssel K_{pub} statt, so kann das Chiffriert nur von dem Teilnehmer mit dem komplementären privaten Schlüssel dechiffriert werden. Auf diese Weise kann dem betreffenden Teilnehmer eine vertrauliche Nachricht zugesandt werden. Findet eine Chiffrierung mit dem privaten Schlüssel K_{priv} statt, so kann das Chiffriert von allen anderen Teilnehmern unter Zuhilfenahme des jeweiligen öffentlichen Schlüssels dechiffriert werden. Auf diese Weise kann die Bestätigung eines Sachverhalts durch einen Teilnehmer erfolgen. Die Bestätigung ist dann auf die genannte Weise durch alle anderen Teilnehmer überprüfbar und nachweisbar. Hierauf beruht die Technik für die Umsetzung **digitaler Signaturen**.

Asymmetrische Chiffrierverfahren werden i.a. zur Vereinbarung geheimer Information während einer Sitzung (Vereinbarung des geheimen Schlüssels), zur Authentisierung im Zusammenhang mit digitalen Zertifikaten und für die digitale Signatur verwendet. Typische Schlüssellängen liegen i.d.R. im Bereich zwischen 512 und 2048 Bit, wobei zurzeit 1024 Bit als hinreichend sicher angenommen wird. Es existieren weitaus weniger asymmetrische als symmetrische Verfahren. Die bekanntesten Verfahren sind Rivest-Shamir-Adleman (RSA), Diffie-Hellman (DH) und Digital-Signing-Algorithm (DSA), wobei für diese Arbeit das RSA-Verfahren im Vordergrund steht.

Asymmetrische Verfahren sind weitaus weniger performant als symmetrische Verfahren. Es besteht hier allerdings nicht die Notwendigkeit einer Schlüsselvereinbarung über einen sicheren Kanal. Ein weiterer Vorteil der asymmetrischen Verfahren ist die Haltbarkeit der Schlüsselpaare. Diese haben i.d.R. eine erheblich längere Gültigkeit (z.T. Jahre) als geheime Schlüssel. Schließlich wächst in einem Kommunikationsnetzwerk die Anzahl der Schlüssel linear mit der Anzahl der Teilnehmer, was durchaus handhabbar ist.

2.3.2.4 Hybride Chiffrierverfahren

Die beiden vorangegangenen Abschnitte haben verdeutlicht, dass sowohl symmetrische als auch asymmetrische Chiffrierverfahren Vor- und Nachteile besitzen. Aus diesem Grund werden sie, wie bereits angedeutet, in unterschiedlichen Weisen angewandt. Die symmetrische Chiffrierung eignet sich aufgrund ihrer hohen Performanz gut für die Wahrung der Vertraulichkeit zwischen zwei Kommunikationspartnern während einer gesamten Sitzung. Der geheime Sitzungsschlüssel, der für eine solche Kommunikation erforderlich ist, wird unter Zuhilfenahme eines asymmetrischen Verfahrens vereinbart. Da dieser Vorgang einmalig für jede Sitzung ist, spielt die Frage nach der Performanz eine unerhebliche Rolle. Diese Strategie wird vom SSL-Protokoll umgesetzt, auf das weiter unten eingegangen werden soll.

2.3.3 Digitale Zertifikate

Genau wie die Kryptographie bilden auch digitale Zertifikate eine wichtige Grundlage dieser Arbeit¹⁷. Digitale Zertifikate nutzen die Kryptographie, um ihren Zweck zu erfüllen. Hierbei kommt insbesondere die asymmetrische Kryptographie zum Einsatz. In den folgenden Abschnitten werden die zentralen Konzepte im Zusammenhang mit digitalen Zertifikaten präsentiert, der wichtigste Zertifikatsstandard eingeführt und die für diese Arbeit besonders wichtigen Zertifikatsspeicher erläutert.

¹⁷ [Oaks01], [TCTrust], [FNMT], [Global], [DPSign]

2.3.3.1 Konzepte

Digitale Zertifikate sind elektronische Artefakte, die das elektronische Pendant zu Ausweisen bilden. Ihre wichtigste Aufgabe ist es, eine digitale Identität mit einem öffentlichen Schlüssel eines bestimmten asymmetrischen, kryptographischen Verfahrens in untrennbarer Weise zu verknüpfen. Hierdurch wird erreicht, dass digitale Handlungen, die die betreffende Identität mit ihrem komplementären, privaten Schlüssel durchführt, auf diese Identität zurückzuführen sind. Mit diesem Ansatz soll die Vertrauenswürdigkeit der digitalen Handlung untermauert werden. Falls nämlich aus dieser Handlung eine evtl. Schädigung folgen sollte, so kann die hierfür verantwortliche Identität festgestellt und zur Rechenschaft gezogen werden. Abb. 6 veranschaulicht das zugrunde liegende Prinzip. Über die hier dargestellten wesentlichen Informationen hinaus können digitale Zertifikate weitere Einträge enthalten, deren Integrität ebenfalls gesichert ist.

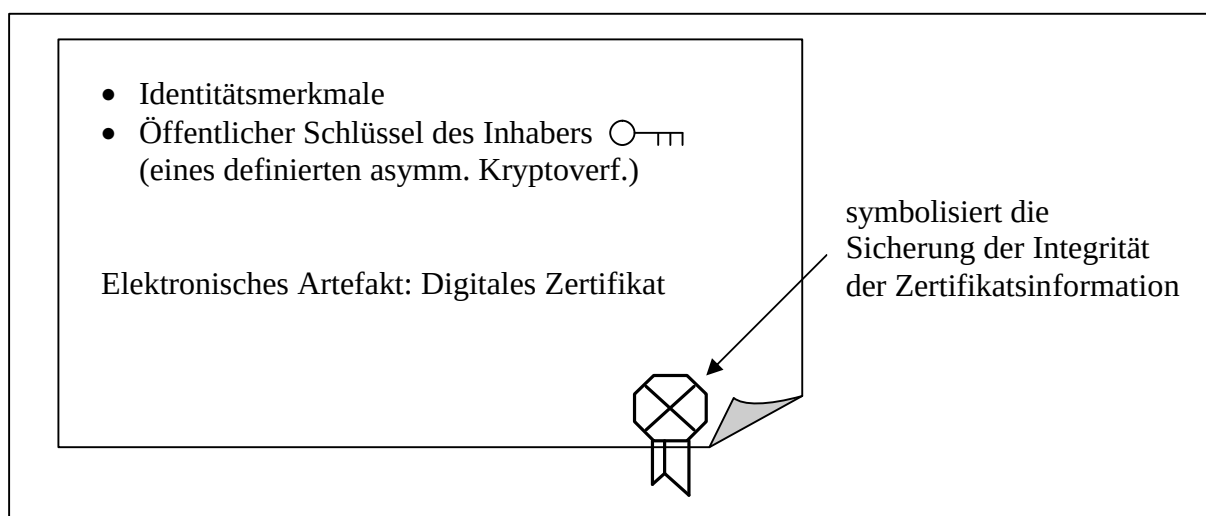


Abb. 6: Elektronisches Artefakt - Digitales Zertifikat

Die Identitätsmerkmale eines digitalen Zertifikats können vielschichtig sein. Die Art der diesbzgl. Einträge werden i.d.R. vom jeweiligen Zertifikatsstandard festgelegt. Innerhalb des jeweiligen Standards können sie zusätzlich vom konkreten Zertifikatszweck abhängig sein.

Der in das Zertifikat eingetragene öffentliche Schlüssel ist einem bestimmten kryptographischen Verfahren zugeordnet. Dieses Verfahren wird i.d.R. durch einen weiteren Eintrag im Zertifikat festgelegt, könnte prinzipiell aber auch durch den Zertifikatsstandard festgelegt werden.

Das in Abb. 6 dargestellte Siegel symbolisiert die Datenintegrität des Zertifikats. Sie wird mit Hilfe asymmetrischer Kryptographie gesichert. Auf diesen Sicherungsvorgang, der einen erheblichen Teil der Komplexität digitaler Zertifikate ausmacht, soll im Folgenden eingegangen werden.

Das Ziel, welches mit dem Einsatz digitaler Zertifikate verfolgt wird, ist die Schaffung von Vertrauen zwischen Kommunikationspartnern in getätigte digitale Handlungen. Ein Teilnehmer vollzieht digitale Handlungen (i.d.R. die digitale Signatur eines Hashwertes, eines Nonce, etc.) mit seinem privaten Schlüssel. Diese Handlungen können von anderen Teilnehmern mit Hilfe des komplementären öffentlichen Schlüssels nachvollzogen und nachgewiesen werden. Einem digitalen Zertifikat kommt dabei die Aufgabe zu, Identitäten und öffentliche Schlüssel untrennbar zu verbinden, damit dies auch für die Identitäten und ihre Handlungen gilt. Grundsätzlich

muss bei jeder Vorlage eines Zertifikats die Frage nach der Vertrauenswürdigkeit des digitalen Zertifikats gestellt werden. Schließlich ist ein vertrauenswürdiges Zertifikat der Garant für eine vertrauenswürdige durchgeführte digitale Handlung. Um die Vertrauenswürdigkeit in digitale Zertifikate herzustellen, existieren übergeordnete Entitäten, die für die Ausstellung der Zertifikate und die Richtigkeit ihrer Daten in gewissem Umfang verantwortlich sind. Diese Entitäten, die Zertifikate ausstellen, werden Zertifizierungsautoritäten (*Certificate Authority*, CA) genannt. Zu den wichtigsten Aufgaben einer CA zählen die Überprüfung der in ein von ihr ausgestelltes digitales Zertifikat eingetragenen Angaben entsprechend der jeweiligen Zertifizierungsrichtlinie und die Sicherung der Datenintegrität der Angaben mit ihrer eigenen digitalen Signatur. Die Zertifizierungsrichtlinie der CA gibt dabei Auskunft darüber, in welchem Umfang die Angaben des Zertifikats vor seiner Ausstellung überprüft werden und ggf. in welcher Weise die CA für die Richtigkeit der Angaben haftet. Zertifizierungsrichtlinien können sehr verschieden sein, so dass dies die qualitative Aussagekraft eines Zertifikats entscheidend prägt. Beispielsweise kann eine einfache Zertifizierungsrichtlinie lediglich die Existenz einer E-Mail-Adresse unter Ausschluss jeglicher Haftung garantieren, während eine umfangreiche Zertifizierungsrichtlinie das persönliche Erscheinen eines Geschäftsführers bei der CA unter Vorlage eines aktuellen Handelsregisterauszugs fordern kann. Unter ein von ihr ausgestelltes Zertifikat leistet die CA mit einem privaten CA-Schlüssel, der charakteristisch für die jeweilige Zertifizierungsrichtlinie ist, eine digitale Signatur, die die Datenintegrität des Zertifikats sichert. Die Datenintegrität wird in Abb. 6 durch das Siegel symbolisiert. Diese Datenintegrität lässt sich anhand des jeweils komplementären öffentlichen Schlüssels der betreffenden Zertifizierungsrichtlinie der CA verifizieren. Abb. 7 zeigt schematisch den beschriebenen Sachverhalt.

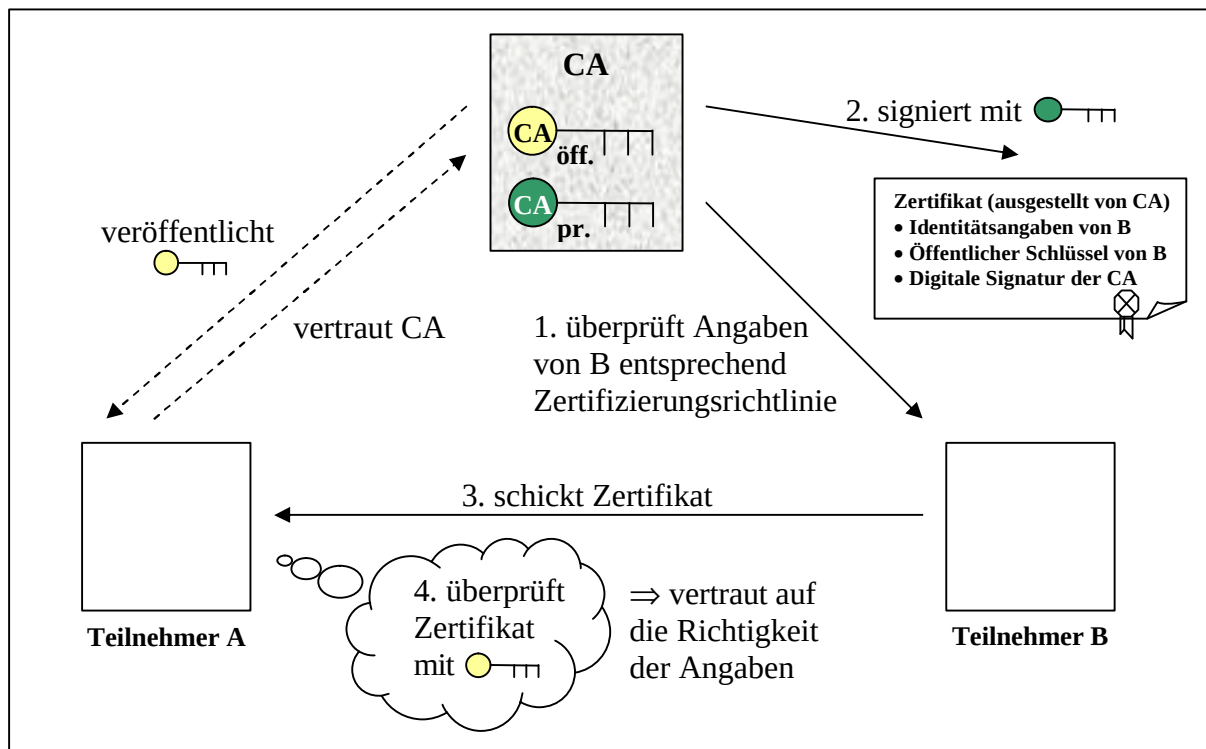


Abb. 7: Integritätsverifizierung von Zertifikaten

Man beachte, dass in Abb. 7 nicht die Vertrauenswürdigkeit von Teilnehmer B geschlussfolgert wird, sondern lediglich die Richtigkeit der Angaben über B. Darüber hinaus wird vorausgesetzt, dass Teilnehmer A der CA vertraut. Nur unter dieser Annahme hat die Schlussfolgerung Bestand. Dieser letzte Punkt führt zu weiteren Diskussionen: Geht man allerdings zunächst davon aus, dass der Teilnehmer A eine erfolgreiche Integritätsverifizierung des Zertifikats durchführen konnte, so kann er anhand der Zertifikatsangaben verifizieren und nachweisen, welche digitalen Handlungen von Teilnehmer B mit dem korrespondierenden privaten Schlüssel vorgenommen worden sind. Die Aussagekraft der Verifikationen und der Nachweise wird hierbei durch die Zertifizierungsrichtlinie bestimmt.

Wie aus dem vorangegangenen Abschnitt hervorgeht, wird die Zertifikatsintegrität mit dem betreffenden öffentlichen Schlüssel der CA festgestellt. Letztlich handelt es sich somit um die Verifikation der digitalen Handlung eines speziellen Teilnehmers, nämlich die der CA. Um dieses Problem zu lösen, sind digitale Zertifikate eingeführt worden. Es hat somit den Anschein, dass die Lösung des Problems das Problem selbst zur Folge hat. Dies ist in gewissen Grenzen richtig. Der öffentliche Schlüssel einer CA bzgl. einer definierten Zertifizierungsrichtlinie wird ebenfalls in Form eines digitalen Zertifikats veröffentlicht. Die Integrität dieses Zertifikats ist folglich mit dem öffentlichen Schlüssel des Ausstellers des CA-Zertifikats überprüfbar, dieser öffentliche Schlüssel befindet sich wiederum in einem Zertifikat, usw. Auf diese Weise erhält man ein hierarchisch strukturiertes Zertifizierungssystem, in dem wenige CA-Zertifikate viele Teilnehmerzertifikate akkreditieren. Dies ist eines der wesentlichen Merkmale eines Systems, das auf digitalen Zertifikaten beruht und das unter dem Sammelbegriff *public key infrastructure (PKI)* zusammengefasst wird.

Das beschriebene Verfahren zeigt, wie man Integritätsschlussfolgerungsketten bilden kann, d.h. wie man aus der Integrität eines CA-Zertifikats die Integrität aller untergeordneter Zertifikate schließen kann. Das Verfahren beruht auf zwei Grundannahmen:

1. Der Teilnehmer, der eine solche Kette verifiziert, vertraut darauf, dass auf jeder Zertifizierungsebene die entsprechende Zertifizierungsrichtlinie in korrekter Weise durchgeführt wird.
2. Der gleiche Teilnehmer vertraut auf die Integrität des CA-Root-Zertifikats (Wurzel- oder Stammzertifikat), auf das er seine Schlussfolgerungen stützt.

Auf den ersten Punkt hat man wenig Einfluss und praktisch keine Möglichkeiten ihn zu überprüfen. Es lässt sich hierfür nur die Plausibilitätserklärung heranziehen, dass die CAs auf allen Ebenen ein eigenes Interesse daran haben, dass das System funktioniert, und sie werden somit dafür Sorge tragen, dass ihre eigenen Richtlinien korrekt eingehalten werden.

Auf den zweiten Punkt hat man hingegen einen entscheidenden Einfluss. Dieser Punkt ist im Wesentlichen dadurch bestimmt, welche Zertifikate man selbst als Teilnehmer zu einem bestimmten Zweck als vertrauenswürdig einschätzt. In der Regel werden dies die Zertifikate oberster Zertifizierungsautoritäten sein. Dies ist allerdings keine Bedingung. Denkbar ist beispielsweise, dass für eine eigene verteilte Anwendung auch eine eigene PKI bereitgestellt wird, weil man

- die Zertifizierungsrichtlinien selbst vorgeben möchte,
- die Zertifikate mit speziellen Merkmalen ausstatten möchte,
- die Zertifikate ausschließlich auf speziellen Speichern bereitstellen möchte (z.B. SmartCards).

In einem solchen Fall wird man vermutlich nur eigene Stammzertifikate als vertrauenswürdig einstufen. Welche Zertifikate jeder Teilnehmer als vertrauenswürdig einschätzt, ist letztlich eine persönliche Ermessensangelegenheit – eine Glaubensfrage. Den Benutzern herkömmlicher Webbrowser wird diese Entscheidung i.d.R. weitestgehend abgenommen, da die Browser bereits mit vorinstallierten, vertrauenswürdigen Stammzertifikaten ausgeliefert werden. Der Durchschnittsnutzer ist sich i.d.R. nicht einmal über die Tragweite einer diesbzgl. Entscheidung im Klaren.

Für erfahrenere Teilnehmer stellt sich allerdings schon die Frage, welche Zertifikate als vertrauenswürdig einzustufen sind und somit den Anker für die Integritätsschlussfolgerungskette bilden sollen. Dies gilt insbesondere für die Serveradministration. Werden Zertifikate oberster CAs als vertrauenswürdig eingeschätzt, so weisen diese Zertifikate i.d.R. ein gemeinsames Merkmal auf. Sie sind selbstsigniert. Selbstsignierte Zertifikate werden mit dem privaten Schlüssel signiert, dessen komplementären öffentlichen Schlüssel sie akkreditieren. Die Integrität eines solchen Zertifikats wird somit nicht mit Hilfe eines anderen Zertifikats verifiziert, sondern mit dem öffentlichen Schlüssel des Zertifikats selbst. Sie sind somit i.d.R. der Ausgangspunkt für eine Integritätsschlussfolgerungskette. Die Integrität eines solchen Zertifikats beruht auf den eigenen Angaben, seine Vertrauenswürdigkeit auf die Überprüfung der Angaben, insbesondere des öffentlichen Schlüssels, durch denjenigen, der ein solches Zertifikat als vertrauenswürdig einzuschätzen hat. Zu diesem Zweck enthalten Zertifikate oftmals ein spezielles Merkmal, das diese Aufgabe erleichtern soll: den *Fingerprint*. Hierbei handelt es sich um den Hashwert des eingetragenen öffentlichen Schlüssels. Diese Hashwerte sind weit kürzer als der öffentliche Schlüssel selbst. Root-CAs veröffentlichen ihre Fingerprints. Derjenige, der zu entscheiden hat, vergleicht die veröffentlichten Fingerprints mit denen in den Zertifikaten. Sind diese identisch, so sind auch die öffentlichen Schlüssel identisch, so dass die Gesamtintegrität des selbstsignierten Zertifikats gewahrt ist. Der Entscheidungsträger hat dann nur noch über die Vertrauenswürdigkeit der Root-CA selbst und ihrer Zertifizierungsrichtlinien zu entscheiden. Selbstsignierte Zertifikate bilden einen Mechanismus, um die Verankerung von Vertrauensbeziehungen herzustellen. Für sich alleine bilden selbstsignierte Zertifikate keinen Vertrauensanker, da ein jeder in der Lage ist, ein selbstsigniertes Zertifikat mit der Identität einer namhaften CA zu erstellen. Lediglich der Abgleich des Fingerprints bietet die erforderliche Sicherheit, dass das selbstsignierte Zertifikat tatsächlich dasjenige der CA ist, das es zu sein vorgibt.

Zusammenfassend kann man sagen, dass i.d.R. jeder Teilnehmer eine Ansammlung von vielen selbstsignierten Zertifikaten besitzt, die er zu unterschiedlichen Zwecken als vertrauenswürdig eingeschätzt¹⁸ hat und die als Vertrauensanker der Integritätsüberprüfung von Zertifikaten der Kommunikationspartner dienen. Zugleich hat jeder Teilnehmer, der selbst digitale Handlungen vornehmen muss, mindestens ein digitales Zertifikat, dessen privaten Schlüssel der Teilnehmer besitzt und das i.d.R. von einer CA fremdsigniert ist.

¹⁸ implizit oder explizit

Zertifikatsrücknahme (*certificate revocation*):

Die bisher dargestellten Sachverhalte beschreiben die wesentlichen Eigenschaften des Paradigmas digitaler Zertifikate. Alle weiteren Eigenschaften im Kontext digitaler Zertifikate sind hiervon abgeleitet oder sekundär. Ähnlich wie reale Ausweise besitzen auch digitale Zertifikate i.d.R. einen Gültigkeitszeitraum. In einem digitalen Zertifikat wird dies durch die Eintragung der Daten, die den Gültigkeitsbeginn und das Gültigkeitsende kennzeichnen, realisiert. Ein Teilnehmer hat also zusätzlich zu der erforderlichen Integritäts- und Vertrauenswürdigkeitsprüfung, die Gültigkeit des Zertifikats zu beachten. Dies ist prinzipiell ein recht simpler Vorgang, da er hierfür lediglich zu vergleichen hat, ob das aktuelle Datum mit Uhrzeit innerhalb der Gültigkeitsspanne liegt. Komplexer kann dieser Vorgang durch ein Konzept werden, das es erlaubt, digitale Zertifikate vor Ablauf des Datums, das das Gültigkeitsende bezeichnet, für ungültig zu erklären. Man bezeichnet diesen Vorgang, der von der CA durchgeführt wird, die das digitale Zertifikat ausgestellt hat, als Zertifikatswiderruf bzw. Zertifikatsrücknahme (*certificate revocation*). Die Möglichkeit, Zertifikate widerrufen zu können, dient dazu, die Sicherheit zusätzlich zu erhöhen. Hierdurch können Zertifikate, die beispielsweise missbräuchlich verwendet wurden, deren Benutzer sich nicht an vereinbarte Vorgaben halten oder die einfach aus einem anderen Grund einzuziehen sind, vorzeitig für ungültig erklärt werden. Die Umsetzung dieses Konzeptes erfolgt mit Hilfe von Listen, in die die widerrufenen Zertifikate eingetragen sind. Man spricht hierbei von Zertifikatswiderrufslisten bzw. Zertifikatsrücknahmelisten (*certificate revocation list*, CRL). In der Praxis sind in diesen Listen nicht die Zertifikate selbst, sondern ihre jeweiligen Seriennummern eingetragen. Dabei existiert für jede Zertifizierungsrichtlinie einer bestimmten CA eine solche Widerrufsliste. Die Zertifizierungsrichtlinie einer CA und eine Zertifikatsseriennummer sind ein eindeutiges Merkmal eines digitalen Zertifikats, so dass der Eintrag der Seriennummer in die CRL ausreichend ist, um ein bestimmtes Zertifikat für ungültig zu erklären. Ein solcher Widerruf ist somit prinzipiell ein recht simpler Vorgang. Die eigentliche Komplexität entsteht durch die Tatsache, dass sich die CRL bei der CA befindet und über die Gültigkeit bei dem jeweiligen Teilnehmer entschieden werden muss. Darüber hinaus muss bei der Entscheidung über die Gültigkeit eines Zertifikats prinzipiell nicht nur eine CRL eingesehen werden, sondern die CRLs sämtlicher Zertifizierungsrichtlinien derer Zertifikate, die die Integritätsschlussfolgerungskette bilden, da prinzipiell ja auch das Zertifikat einer CA für ungültig erklärt werden könnte und dies die Ungültigkeit sämtlicher Unterzertifikate zur Folge hätte. Zwei Probleme sind somit von zentraler Bedeutung: die Aktualität einer Vielzahl von CRLs und ihre jeweilige Authentizität. Die Authentizität einer CRL wird ähnlich der Integrität eines digitalen Zertifikats durch eine digitale Signatur erreicht. Der Benutzer einer CRL kann sich durch Prüfung der Signatur davon überzeugen, dass die CRL tatsächlich von der betreffenden CA ausgestellt worden ist und dass ihr Inhalt unverfälscht ist. Für die Aktualität einer CRL gibt es prinzipiell zwei Strategien. Bei der ersten Variante wird die CRL als lokale Datei gehalten und muss dementsprechend in regelmäßigen Abständen¹⁹ aktualisiert werden. Dies kann natürlich von einem eigenen Prozess automatisiert werden. Bei der zweiten Variante wird bei Bedarf eine Anfrage an einen CRL-Server der CA geschickt. Beide Strategien haben jeweils Vor- und Nachteile. Aus diesem Grund empfiehlt sich der Einsatz einer Mischlösung, um diesbzgl. den höchsten Sicherheitsgrad zu erreichen. Da CRLs für die vorliegende Arbeit eine untergeordnete Rolle spielen, soll hierauf nicht weiter eingegangen werden.

¹⁹ Richtwert: wöchentlich

2.3.3.2 X.509 – Ein Standard für digitale Zertifikate

Im vorangegangenen Abschnitt sind die Konzepte digitaler Zertifikate in allgemeiner Weise beschrieben worden. Die Betrachtungen in diesem Abschnitt konzentrieren sich auf einen speziellen Zertifikatsstandard – dem X.509 Standard. Obwohl natürlich auch weitere Standards digitaler Zertifikate existieren, wie beispielsweise der Pretty-Good-Privacy Standard (PGP), ist der X.509 Standard am weitesten verbreitet und zugleich auch der einzig unterstützte Standard von der verwendeten Programmierumgebung²⁰. Vor allem aus dem letzten Grund ist er für diese Arbeit der einzig bedeutungsvolle.

Der X.509 Standard existiert seit 1988 in der Version 1 (X.509v1). Der z.Z. gebräuchliche Industriestandard ist der X.509v3 Standard. Er weicht nur geringfügig von der ersten Version ab. In der dritten Version ist es möglich, die Zertifikate durch sog. Erweiterungen (*extensions*) an individuelle Bedürfnisse anzupassen. Diese Möglichkeit wird beispielsweise dazu verwendet, die Verwendung von Zertifikaten einschränken zu können, ähnlich dem Vorbild des tangiblen Ausweises (Personalausweis, Führerschein, Studentenausweis, usw.). Die Möglichkeiten, die neuere Versionen für diese Arbeit bieten könnten, sind allerdings zunächst von geringer Bedeutung, da zunächst der Identitätsnachweis des Zertifikatsinhabers im Mittelpunkt der Betrachtung steht. Darüber hinaus bietet die verwendete Programmierumgebung lediglich die Möglichkeit²¹, selbstsignierte X.509v1 Zertifikate zu erstellen, die zudem nicht in gängige Browser wie Netscape exportierbar sind.

Außer den genannten Extensions ist die wichtigste Eigenschaft dieses Zertifikatsstandards ergänzend zu den generischen Zertifikatskonzepten die Art, wie Identitätsmerkmale festgelegt werden. Diese Identitätsmerkmale sind in einem sog. *X.500 Distinguished Name (DN)* zusammengefasst. Auf Details des X.500 Standards zur Strukturierung globaler Verzeichnisse²² soll hier nicht eingegangen werden, da sie für diese Arbeit nicht von Bedeutung sind. Die einfachste Art, einen solchen X.500 DN zu verstehen, ist ihn als geordnete Ansammlung von Variablen zu sehen, denen Werte zugeordnet sein können. Die Variablen repräsentieren Identitätsmerkmale, die in ein X.509 Zertifikat aufgenommen werden können. Die Tab. 1 gibt ein Beispiel, welche Variablen üblicherweise vorkommen:

Variable	Inhalt	Erläuterung
b	E-Mail-Adresse	die E-Mail-Adresse der Identität
CN	Common Name	der volle Name der zertifizierten Identität
OU	Organizational Unit	die Organisationseinheit, der die Identität zugeordnet wird
O	Organization	die Organisation, für die die Identität tätig ist
L	Location	der Ort, in dem sich die Identität übl. befindet
S	State	die Region, in dem sich der Ort befindet
C	Country	der Staat, dem die Region zugeordnet ist

Tab. 1: Identitätsmerkmale in X.509 Zertifikaten

²⁰ J2SDK 1.4.1.

²¹ mittels des Zertifikatsmanagementwerkzeugs *keytool*

²² www.webopedia.com/TERM/X/X_500.html

Welche Variablen tatsächlich vorkommen und somit auch Werte haben, hängt von der jeweiligen Zertifizierungsrichtlinie ab. Ein X.509 Zertifikat enthält zwei X.500 DN's: ein DN für den Aussteller (*certificate issuer*), der die Identitätsmerkmale der CA inkl. der jeweiligen Zertifizierungsrichtlinie wiedergibt, und ein DN für den Inhaber (*certificate subject*), dessen Identität durch das Zertifikat akkreditiert wird. Bei selbstsignierten Zertifikaten sind beide DN's identisch.

Für diese Arbeit ist letztlich wichtig, dass es einen Zertifikatsstandard gibt, der von der verwendeten Zielumgebung unterstützt wird und dessen Dateneinträge im Rahmen einer Richtlinie als gesichert gelten können. Der X.509 Standard ist ein solcher Standard.

2.3.3.3 Zertifikatsspeicher in Java – serverseitige Betrachtung

In Abschnitt 2.2 ist bereits erläutert worden, dass die Zielplattform serverseitig aus einem javabasierten Application Server besteht. In Abschnitt 2.3.4.3 wird darüber hinaus die wesentliche Möglichkeit für die Implementierung des SSL-Protokolls in Java erläutert – die Nutzung der *Java Secure Socket Extension* (JSSE, s. 2.3.4.3). Innerhalb der JSSE wird u.a. mit dem Zweck der gegenseitigen Authentisierung auf Zertifikate zurückgegriffen. Der zu diesem Zweck in Java bereitgestellte Speichermechanismus soll deshalb in diesem Abschnitt dargestellt werden.

Zertifikatsspeicher werden in Java als sog. **Keystores** bereitgestellt. Hierbei sind zwei wesentliche Umgangsformen mit Keystores zu nennen: administrativ und programmatisch. Für den administrativen Umgang werden Keystores üblicherweise als Keystore-Dateien gehalten. Diese werden mit einem eigenständigen Programm namens *keytool* verwaltet, das im Verzeichnis *\$java_home\$/bin* des J2SDK zu finden ist. Programmatisch werden Keystores durch die Klasse *java.security.KeyStore* verkörpert. Diese können durch das Laden aus einer Keystore-Datei instanziiert werden. Selbstverständlich ist auch das Sichern einer KeyStore-Instanz in eine Datei möglich.

Die Klasse KeyStore ist eine Engine-Klasse (s. JSA). Engine-Klassen definieren eine API für eine Softwarekomponente, so dass verschiedene Implementierungen für die jeweilige Komponente existieren können. Die J2SDK v1.4. stellt für Keystores drei Implementierungen zur Verfügung: Java KeyStore (JKS), JCE KeyStore (JCEKS) und PKCS#12. Auf die letzte Implementierung kann allerdings nur lesend zugegriffen werden.

Unabhängig von der betreffenden Umgangsform haben Keystores die Datenstruktur eines *Dictionaries*, d.h. dass auf die in einem Keystore abgelegten Einträge über ein Schlüsselobjekt zugegriffen wird. Bei Keystores sind die Schlüsselobjekte einfache Namen vom Typ *String*. Sie werden als *Alias* bezeichnet. Die Einträge werden in zwei Kategorien gegliedert: Schlüsseleinträge (*key entries*) und Zertifikatseinträge (*certificate entries*). Key entries enthalten Schlüssel – geheime (symmetrische) Schlüssel oder private (asymmetrische) Schlüssel. Bei einem asymmetrischen Schlüssel enthält der Eintrag zusätzlich das korrelierte, öffentliche Zertifikat bzw. eine ganze Kette öffentlicher Zertifikate, an dessen Anfang das korrelierte, öffentliche Zertifikat steht und dem die jeweiligen CA-Zertifikate folgen. Certificate entries hingegen enthalten lediglich ein einzelnes, öffentliches Zertifikat ohne den korrelierten, privaten Schlüssel.

Die Zertifikate, die als key entries oder als certificate entries gehalten werden, haben hierbei unterschiedliche Aufgaben. Bei den Zertifikaten, die als key entries gehalten werden, handelt es sich um eigene Zertifikate, die dazu dienen, eigene digitale Handlungen zu verrichten. Deshalb werden sie auch zusammen mit dem jeweils korrelierten, privaten Schlüssel gehalten. Bei den Zertifikaten, die als certificate entries gehalten werden, handelt es sich um Zertifikate, die als vertrauenswürdig eingestuft worden sind. I.d.R. sind dies selbstsignierte Stammzertifikate der CAs.

Wie bereits erwähnt werden Keystore-Dateien mit einem Programm namens keytool verwaltet. Hierbei handelt es um eine Befehlszeilenorientierte Anwendung (kein GUI). Der Umgang mit diesem Programm ist relativ umständlich und für ernsthafte Verwaltungsaufgaben nicht zu empfehlen. Dennoch ist es möglich, die meisten erforderlichen Aufgaben mit Hilfe von keytool zu bewältigen. So ist es möglich, asymmetrische Schlüsselpaare und korrelierte selbstsignierte Zertifikate zu generieren (X.509 v1), Zertifikatssignieranträge an CAs aus selbstsignierten Zertifikaten (*Certificate Signing Request*, *CSR*) zu erstellen, Zertifikate (eingeschränkt) zu importieren und zu exportieren und vieles mehr. Für eine ausführliche Darstellung der Befehlszeilenkommandos wird auf die Onlinedokumentation²³ und auf die JSA-Literatur verwiesen.

In dieser Arbeit werden die Keystores im Kontext der JSSE benötigt (s. 2.3.4.3). Die einfachste Möglichkeit einem JSSE-Server mitzuteilen, welche Zertifikate er verwenden soll, ist über Java-Systemeigenschaften (*System Properties*). Hierbei wird die entsprechende Eigenschaft mit dem vollständigen Keystore-Dateinamen initialisiert. Bei dieser Variante darf das Keystore, das das Serverzertifikat und den korrelierten privaten Schlüssel enthält, nur diesen einen key entry enthalten. Erfordert der Server eine Client-Authentisierung, wird die Initialisierung einer zweiten Systemeigenschaft benötigt. Dies geschieht ebenfalls mit dem vollständigen Keystore-Dateinamen der Keystore-Datei, die sämtliche für den betreffenden Server relevanten, vertrauenswürdigen Zertifikate enthält. Die erste Datei bildet somit den *JSSE-Keystore*, d.h. den Speicher für Zertifikate, die zum Nachweis der eigenen Identität dienen, und die zweite den *JSSE-Truststore*, d.h. den Speicher für Zertifikate, die als Vertrauensanker zugelassen sind. Die Namensgebung ist in diesem JSSE-Kontext leider etwas irritierend, da beide Dateien jeweils durch ein Keystore implementiert werden.

2.3.3.4 Zertifikatsspeicher in Netscape – clientseitige Betrachtung

Wie bereits in Abschnitt 2.2 erwähnt, liegen clientseitig Webbrowser vor, wobei für die Erprobung des programmatischen Teils dieser Arbeit Netscape 7.0 verwendet worden ist. Dies ist insbesondere in Hinblick auf die Nutzung der verwendeten SmartCard von Bedeutung (s. 2.3.5). Wird ein anderer Browser verwendet, so wird der betreffende Zertifikatsspeicher des Browsers konzeptionell eine ähnliche Struktur aufweisen. Um die Anbindung des SmartCard-Lesegeräts an einen anderen Browser zu bewerkstelligen, ist darauf zu achten, dass der betreffende Browser eine der SmartCard-Schnittstellen unterstützt.

²³ [J2Sun02] → Security Tool → keytool, siehe auch [Oaks01]

Netscape 7.0 speichert Zertifikate und ggf. korrelierte private Schlüssel in sog. **Sicherheitseinrichtungen**²⁴. Eine Sicherheitseinrichtung ist somit der physikalische Speicher für Zertifikate und Schlüssel. Die Standard-Sicherheitseinrichtung von Netscape ist der sog. **Software-Sicherheitsdienst**, der eine Speicherung eigener Zertifikate auf der Festplatte in einer nicht näher erwähnten Datei vornimmt. Ein weiteres Beispiel für eine Sicherheitseinrichtung ist ein SmartCard-Lesegerät in Kombination mit den zu verwendenden SmartCards. Allgemein stellen Sicherheitseinrichtungen kryptographische Dienste zur Verfügung, so dass es sich hierbei nicht zwingend um Speicher handeln muss. Der Zugriff auf jede Sicherheitseinrichtung ist durch ein sog. **Master-Passwort** geschützt. Verwaltet werden die Sicherheitseinrichtungen von Netscape mit Hilfe des **Geräte-Managers**²⁵. Dieser zeigt sämtliche vorhandenen Sicherheitseinrichtungen an und erlaubt, neue hinzuladen bzw. vorhandene zu entladen. Hierbei sind die einzelnen Sicherheitseinrichtungen stets einem sog. **PKCS#11-Modul** untergeordnet. Ein solches Modul ist ein Plug-In und ist gewissermaßen als Treiber für die Sicherheitseinrichtung unter Netscape zu verstehen. Kryptogeräte, die unter Netscape als Sicherheitseinrichtungen betrieben werden sollen, müssen zwingend die PKCS#11-API²⁶ unterstützen und ein entsprechendes Modul (Plug-In) zur Verfügung stellen. Die Zertifikate, die in registrierten Sicherheitseinrichtungen gespeichert sind, können mit Hilfe des **Zertifikats-Managers** verwaltet werden. Abb. 8 zeigt den dargestellten Sachverhalt schematisch. Eine ausführliche Darstellung über den Umgang mit Zertifikaten unter Netscape kann der Netscape Programm-Dokumentation entnommen werden ([Nets02], Einstellungen → Zertifikate → Hilfe).

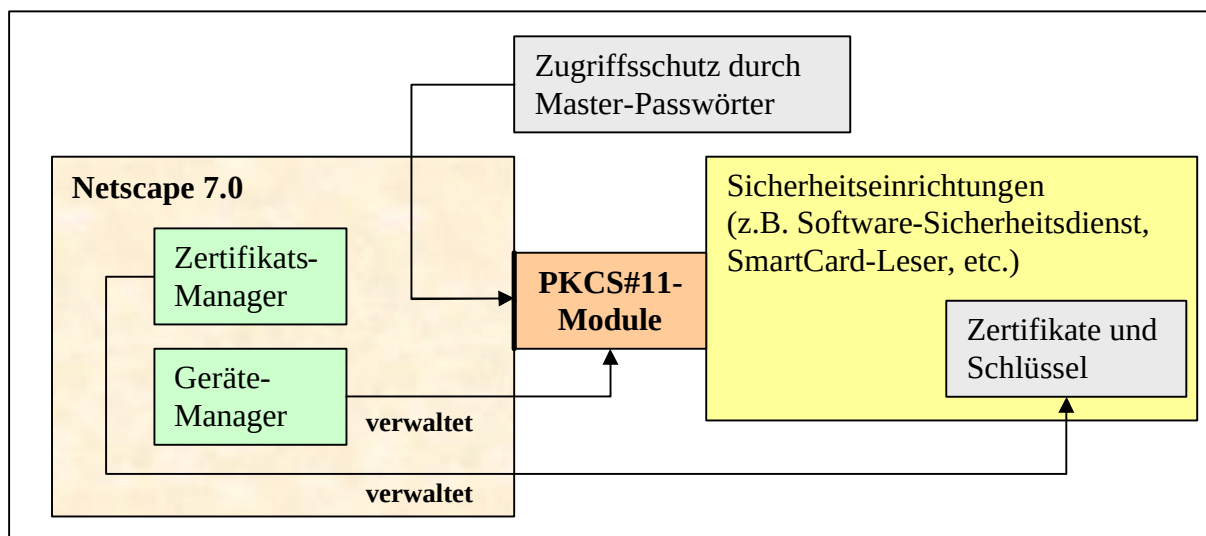


Abb. 8: Zertifikatsspeicher in Netscape - Sicherheitseinrichtungen

2.3.4 Das SSL-Protokoll

Das **Secure Socket Layer (SSL)** Protokoll ist innerhalb dieser Arbeit die Schlüsseltechnologie, mit dessen Hilfe ein beachtlicher Teil der Sicherheitskonzepte umgesetzt werden. Die meisten der bisher erläuterten Technologien und Mechanismen sind Basistechnologien, die im SSL-Protokoll wieder verwendet werden.

²⁴ auch als (Sicherheits-)Token bezeichnet

²⁵ dieser ist über den Menüpunkt *Einstellungen* erreichbar (gilt auch für Zertifikats-Manager)

²⁶ auch als Cryptoki (**C**ryptographic **T**oken **I**nterface) bezeichnet

2.3.4.1 Der Protokollstapel – eine statische Betrachtung

In diesem Abschnitt soll das SSL-Protokoll zunächst aus statischer Sicht betrachtet werden, um es in Hinblick auf eine Internetsicherheitsarchitektur einzuordnen.

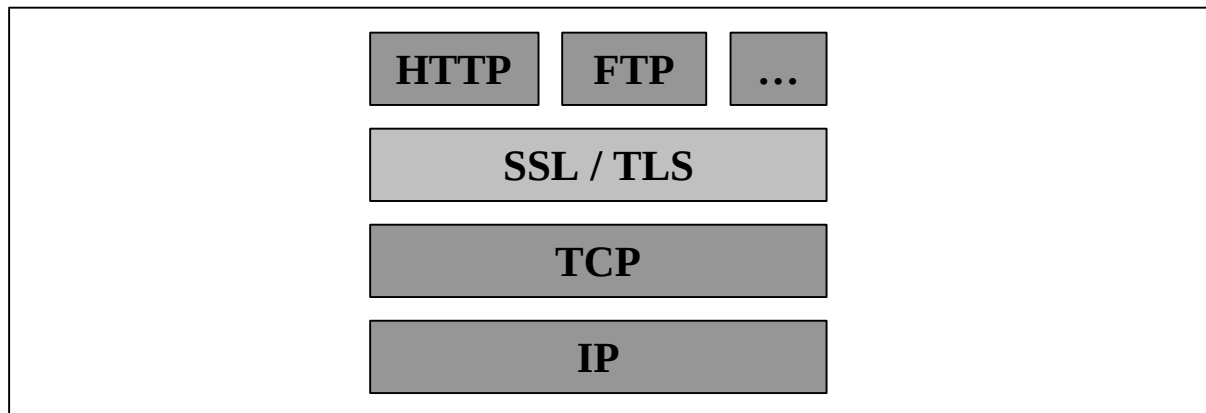


Abb. 9: SSL-Protokollstapel

Abb. 9 zeigt das SSL-Protokoll zwischen den TCP/IP-Schichten und den jeweiligen Internetanwendungsprotokollen. Das SSL-Protokoll wurde vom Browserhersteller Netscape entwickelt und hat sich als defacto-Standard für Kommunikationssicherheit im Internet etabliert. Die aktuelle Version ist SSLv3. Nach der Übernahme des Standards durch die IETF²⁷ wurde SSL in **Transport Layer Security (TLS)** umbenannt. Die aktuelle Version TLSv1 ist weitestgehend mit SSLv3 identisch.

Wie erwähnt, dient SSL zur Bereitstellung von Kommunikationssicherheit. Dies impliziert die in 2.1.2.3 und 2.1.2.4 erläuterten Sicherheitskonzepte Vertraulichkeit und Integrität der kommunizierten Daten. Zusätzlich besteht die Möglichkeit, das Konzept Authentisierung (s. 2.1.2.1) mit Hilfe von SSL umzusetzen. Die Vielzahl der von SSL umgesetzten Konzepte verdeutlicht die zentrale Bedeutung dieser Technologie für diese Arbeit. Um die genannte Funktionalität bereitzustellen, bedient sich SSL der Einweg-Hashfunktionen, der symmetrischen und asymmetrischen Kryptographie und der digitalen Zertifikatsarchitektur. Dies wiederum stellt den Bezug zu zuvor erläuterten Technologien her.

Das SSL-Protokoll arbeitet hierbei transparent für die übergeordneten und die untergeordneten Protokolle. Dies bedeutet, dass ein Protokoll aus der Anwendungsschicht (http, ftp, usw.) keinen Unterschied darin sieht, ob es direkt mit dem TCP-Protokoll oder mittelbar über das SSL-Protokoll kommuniziert. Umgekehrt nimmt das TCP-Protokoll keine Unterschiede zwischen einer direkten Kommunikation mit dem betreffenden Anwendungsprotokoll und einer mittelbaren über das SSL-Protokoll wahr. Man beachte hierbei, dass zwar ein beliebiges Anwendungsprotokoll verwendbar ist, dass aber SSL stets TCP voraussetzt. UDP und evtl. andere Protokolle der Transportschicht schließen die Verwendung von SSL aus.

Oft wird SSL im Kontext eines *Secure Web Servers* verwendet. Dies meint, die Nutzung von *http über ssl*, was auch oft zusammenfassend als *https* bezeichnet wird. Der Standardport dieses zusammengefassten Protokolls (https) ist 443. Üblicherweise findet bei https eine Authentisierung des Web Servers durch den Clientbrowser statt. Hierfür stellt der Server ein SSL-Serverzertifikat bereit, das von gängigen Browsern akzeptiert wird, d.h. ein Zertifikat, welches vom Browser als

²⁷ IETF: Internet Engineering Task Force, www.ietf.org

vertrauenswürdig anerkannt wird. Prinzipiell ist es zwar auch möglich, ein Zertifikat zu verwenden, welches nicht als vertrauenswürdig anerkannt wird (eine SSL-Verbindung kann hiermit auch aufgebaut werden), der Benutzer wird aber in diesem Fall mit einer Vielzahl von Warnungen konfrontiert werden, dessen Tragweite er i.d.R. nicht zu beurteilen vermag. Der komplementäre Vorgang der Clientauthentisierung findet bei SSL hingegen üblicherweise nicht statt.

2.3.4.2 Der SSL-Handshake – eine dynamische Betrachtung

Im vorangegangenen Abschnitt hat eine Einordnung des SSL Protokolls in den Protokollstapel stattgefunden und es wurde konstatiert, dass durch die Zwischenschaltung dieser Sicherheitsschicht viele Sicherheitskonzepte automatisch umgesetzt werden. Auf diesen letzten Punkt soll in diesem Abschnitt genauer eingegangen werden, in dem gezeigt wird, wie das SSL Protokoll die genannten Sicherheitskonzepte (s. 2.3.4.1) umsetzt. Eine wichtige Rolle für das Verständnis der Umsetzung spielt hierbei der *Handshake*, also derjenige Teil des Protokolls, der für den Aufbau der Verbindung verantwortlich ist. Er soll deshalb hier detailliert beschrieben werden²⁸:

1. **Client-Hello:** Der Client schickt dem Server einen zufälligen *Challenge-String* und benennt diejenigen Algorithmen, die er für die Kommunikation bzw. ihren Aufbau verwenden möchte (Abb. 10). Die Gesamtheit der Algorithmen bezeichnet man als *Cipher-Suite*. Eine Cipher-Suite besteht aus einem Algorithmus zur Schlüsselvereinbarung (asymmetrisches Chiffrierverfahren, z.B. RSA, DH, etc.), einem Verschlüsselungsalgorithmus (symmetrisches Chiffrierverfahren, z.B. DES, RC4, etc.) und einer Einweg-Hashfunktion (z.B. SHA, MD5, etc.).
2. **Server-Hello:** Der Server beantwortet die Anfrage mit einer Bestätigung der Cipher-Suite. Zusätzlich schickt er dem Client das Serverzertifikat und eine Verbindungsidentifikationsnummer (*connection identifier, CID*) (Abb. 10).

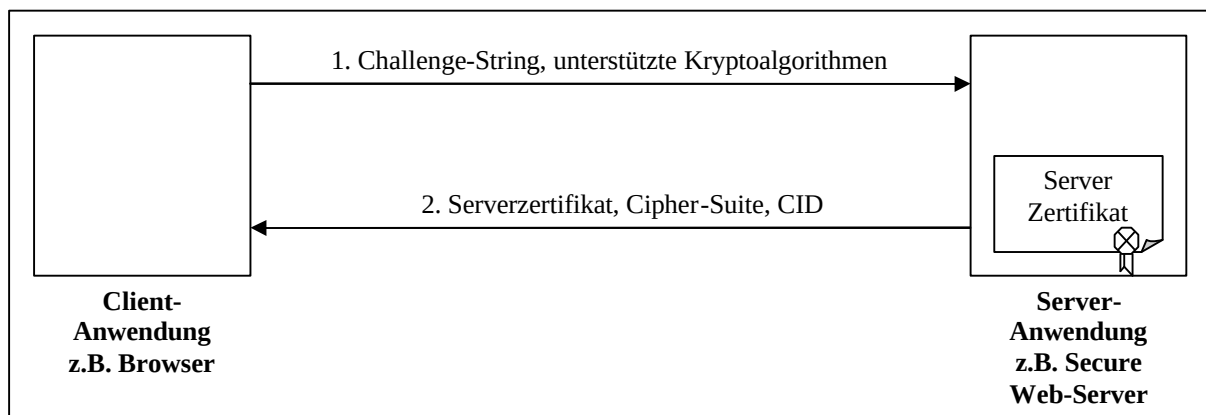


Abb. 10: SSL-Handshake (Client- und Server-Hello)

3. **Gültigkeitsprüfung:** Der Client überprüft die Gültigkeit des Serverzertifikats. Dies beinhaltet insbesondere die Prüfung der Zertifikatssignatur (Abb. 7, Abb. 11), ggf. aber auch die weiteren erforderlichen Prüfungen wie Datum, CRL, etc. (vgl. 2.3.3.1).

²⁸ [VeriSign98], www.verisign.com/repository/clientauth/ent_ig.htm

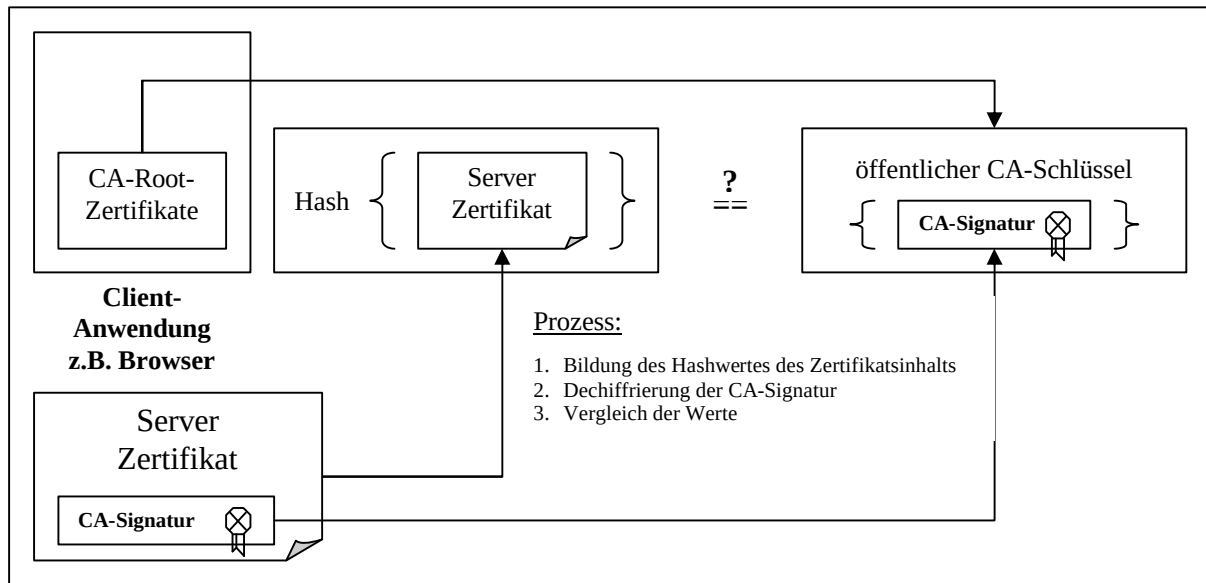


Abb. 11: SSL-Handshake (Clientseitige Signaturprüfung)

Das vom Server vorgelegte Zertifikat muss von einer dem Client vertrauenswürdig erscheinenden CA signiert worden sein. Eine CA wird als vertrauenswürdig eingestuft, wenn ihr Zertifikat oder das einer übergeordneten CA im Browser installiert worden ist (vgl. 2.3.3.1). Die Signatur ist der mit dem privaten Schlüssel der CA asymmetrisch chiffrierte Hashwert des Zertifikatsinhalts. Dementsprechend wird zur Signaturprüfung die Signatur mit dem öffentlichen Schlüssel der CA dechiffriert und der Hashwert des Zertifikats gebildet. Stimmen beide Werte anschließend überein, so ist die Schlussfolgerung, dass das vorgelegte Serverzertifikat tatsächlich von der CA unterschrieben worden ist, d.h. dass es nicht manipuliert worden ist. Hieraus folgt noch nicht die Server-Authentisierung, da das vorgelegte Serverzertifikat öffentlich ist und somit im Prinzip von jedem vorgelegt werden kann. Ist auf dem Client kein vertrauenswürdiges Zertifikat installiert, mit dem eine Integritätsprüfung stattfinden könnte, so wird der Benutzer am Client aufgefordert, die Sitzung abzubrechen, das vorgelegte Zertifikat für die anstehende Sitzung zu akzeptieren oder das vorgelegte Zertifikat als stets vertrauenswürdig zu akzeptieren. Damit der Vorgang fortgesetzt wird, muss sich der Benutzer für eine der letzten beiden Möglichkeiten entscheiden. Ansonsten kommt die Verbindung nicht zustande.

4. **Client Master Key:** Der Client erzeugt einen Hauptsitzungsschlüssel (*master session key*), chiffriert ihn asymmetrisch mit dem öffentlichen Schlüssel des Servers, welcher aus dem Serverzertifikat entnommen wird, und schickt dem Server das Chifftrat (Abb. 12). Da der Zertifikatsinhaber (nämlich der zertifizierte Server) als einziger im Besitz des zum Zertifikat zugehörigen privaten Schlüssels ist, ist er auch als einziger in der Lage, das Chifftrat des Hauptsitzungsschlüssels zu dechiffrieren.

Der clientseitig erzeugte Hauptsitzungsschlüssel dient beiden Seiten zur Erzeugung von Schreib- und Leseschlüsseln (s. 5).

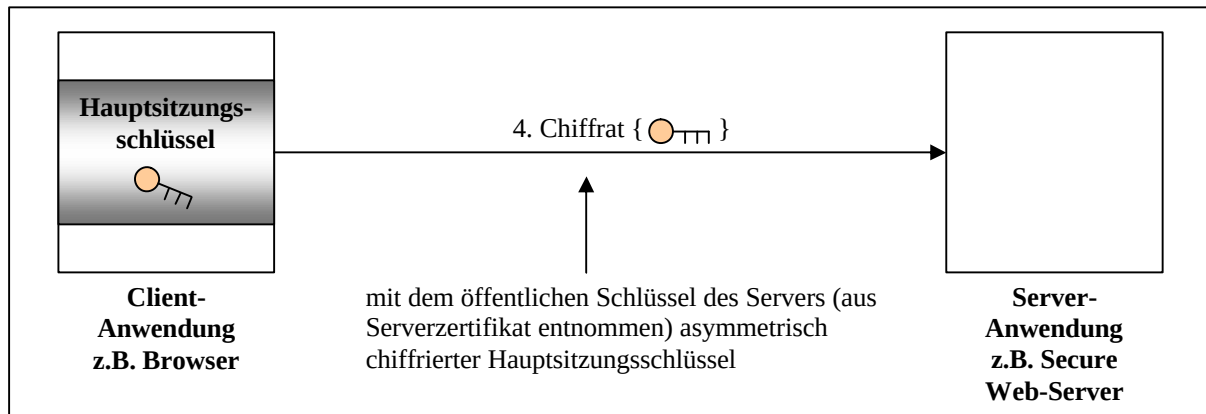


Abb. 12: SSL-Handshake (Hauptsitzungsschlüssel)

5. **Sitzungsschlüsselerzeugung:** Der Client ermittelt aus dem erzeugten Hauptsitzungsschlüssel zwei Sitzungsschlüssel: einen Schlüssel zum Schreiben (*ClientWriteKey*) und einen Schlüssel zum Lesen (*ClientReadKey*) (Abb. 13). Der Server dechiffriert zunächst das empfangene Chiffprat mit seinem privaten Schlüssel und ermittelt so den Hauptsitzungsschlüssel. Anschließend führt er die gleiche Prozedur wie der Client durch. Auf diese Weise erzeugt er auch zwei Sitzungsschlüssel: einen Schlüssel zum Lesen (*ServerReadKey*) und einen Schlüssel zum Schreiben (*ServerWriteKey*) (Abb. 13). Man beachte, dass der *ClientWriteKey* und der *ServerReadKey* identisch sind (symmetrische Verschlüsselung). Gleiches gilt für den *ClientReadKey* und den *ServerWriteKey*.

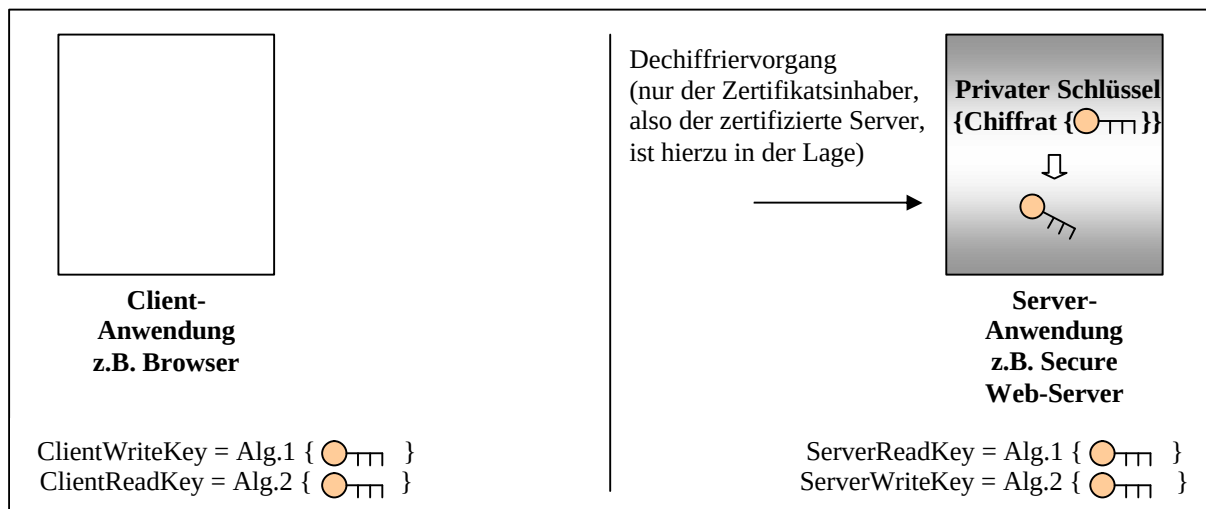


Abb. 13: SSL-Handshake (Schreib- und Leseschlüssel)

6. **Client Finish:** Der Client schickt dem Server ein Acknowledgement und wartet auf die Bestätigung durch den Server. Hierzu wird die CID aus 2. mit dem *ClientReadKey* symmetrisch verschlüsselt und an den Server geschickt. Da *ClientReadKey* und *ServerWriteKey* identisch sind, ist dem Server bekannt, dass der Client in der Lage ist, die Nachrichten des Servers zu entschlüsseln, wenn beim Entschlüsseln mit dem *ServerWriteKey* tatsächlich die CID herauskommt.
7. **Server Verify:** Der Server antwortet dem Client mit dem durch den *ServerWriteKey* symmetrisch verschlüsselten Challenge-String aus 1., den der Client mit seinem *ClientReadKey* entschlüsselt. Ist der entschlüsselte Challenge-String mit dem aus 1. identisch, so schlussfolgert der Client, dass der Server den *ServerWriteKey* ermitteln konnte (Abb. 14). Aus der Kenntnis auf Seiten des

Servers des ServerWriteKey folgert der Client weiter, dass der Server hierzu vorher den Hauptsitzungsschlüssel hat ermitteln müssen. Da dies aus der übermittelten Information nur mit Hilfe des zum Serverzertifikat zugehörigen privaten Schlüssels möglich ist, schließt der Client hieraus, dass sein Kommunikationspartner im Besitz des zum vorgelegten Zertifikat zugehörigen privaten Schlüssels ist. Hieraus folgert der Client die Authentizität des Servers.

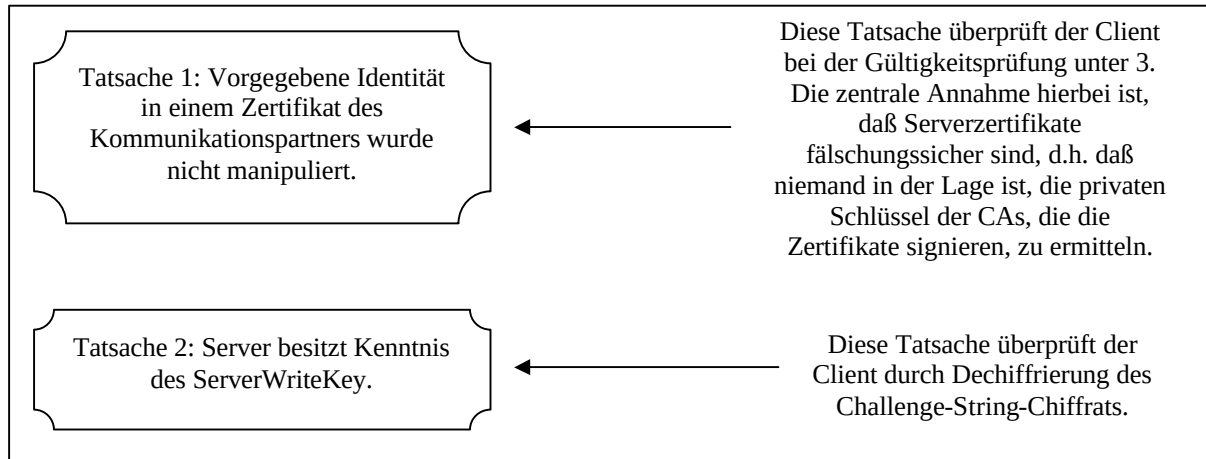


Abb. 14: SSL-Handshake (Server Verify 1/2)

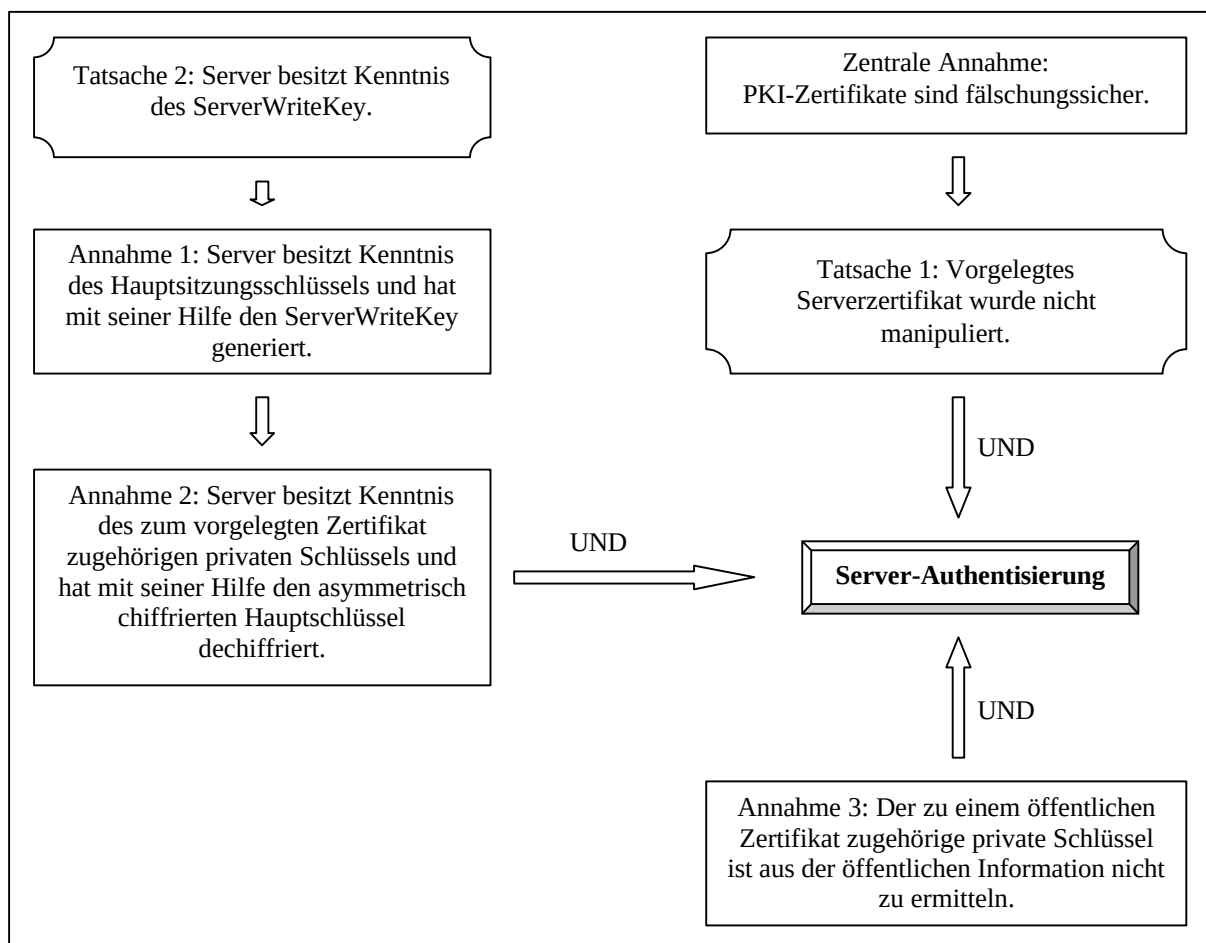


Abb. 15: SSL-Handshake (Server Verify 2/2)

Fazit 1: Die Vertraulichkeit der Kommunikation ist durch das geheime Schlüsselpaar gewährleistet. Die Integrität der Kommunikation ist durch die mögliche Bildung von Hashwerten gewährleistet. Die Authentizität des Servers wird wie in 7. beschrieben geschlussfolgert.

8. **Request Certificate:** Ist der Server für eine Client-Authentisierung konfiguriert, so schickt er dem Client chiffriert einen neuen Challenge-String (NeuerChallengeString). Der Client antwortet mit einer Fehlermeldung, falls er sich nicht authentisieren kann oder will (Abb. 16).

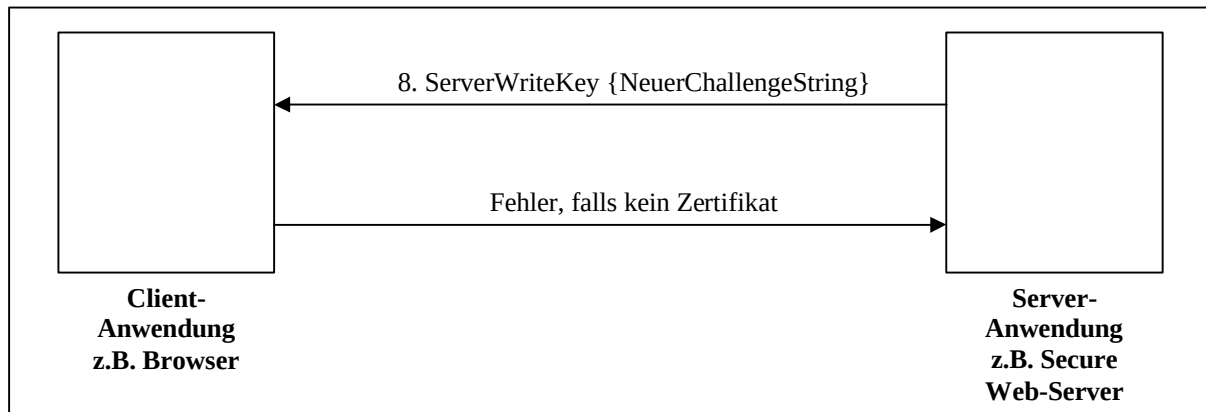


Abb. 16: SSL-Handshake (Request Certificate)

9. **Client Certificate (Schritt 1):** Ist der Client in der Lage ein gültiges Zertifikat vorzulegen, so ermittelt dieser zunächst den neuen Challenge-String. Ferner ermittelt er einen Hashwert, bestehend aus dem ermittelten neuem Challenge-String und dem in 2. empfangenem Serverzertifikat. Diesen signiert der Client mit seinem privaten Schlüssel. Man beachte, dass durch das Einbringen des neuen Challenge-Strings die Signatur des Hashwertes bei unterschiedlichen Sitzungen stets unterschiedliche Werte annimmt.

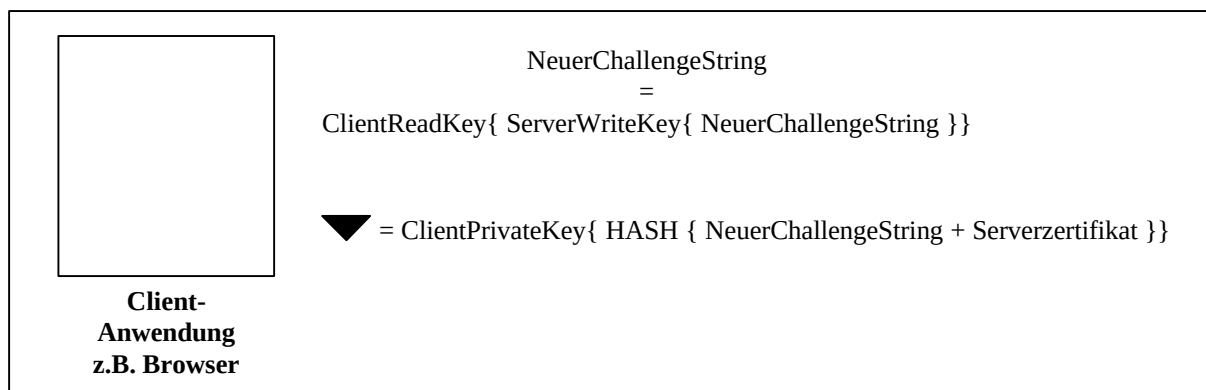


Abb. 17: SSL-Handshake (Client Certificate 1/3)

10. **Client Certificate (Schritt 2):** Der Client schickt dem Server verschlüsselt den signierten Hashwert und sein Clientzertifikat (Abb. 18).

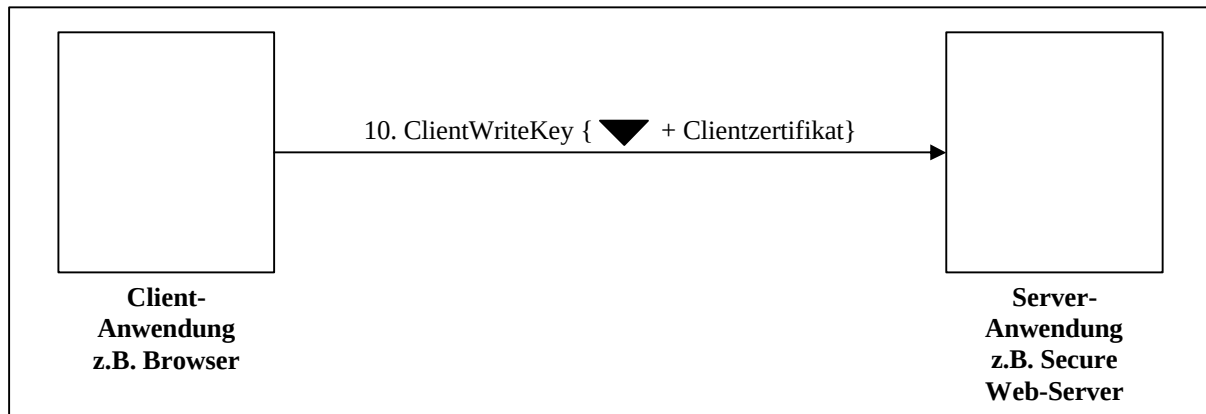


Abb. 18: SSL-Handshake (Client Certificate 2/3)

11. Client Certificate (Schritt 3): Der Server ermittelt den signierten Hashwert und das Clientzertifikat aus dem empfangenen Chifftrat. Er überprüft die Gültigkeit des Zertifikats anhand des öffentlichen Schlüssels der CA (siehe hierzu auch 3.). Um sich zu vergewissern, dass der Zertifikatsbesitzer (Client) auch tatsächlich der Zertifikatsinhaber (Instanz, für die das Zertifikat ausgestellt worden ist), dechiffriert der Server die Signatur des Hashwertes mit dem öffentlichen Schlüssel aus dem Clientzertifikat und bildet seinerseits den Hashwert aus neuem Challenge-String und Serverzertifikat. Stimmen beide Werte überein, schlussfolgert der Server die Authentizität des Clients (Abb. 19).

Man beachte, dass im Gegensatz zum Server der Client die Kenntnis des zum vorgelegten Zertifikat zugehörigen privaten Schlüssels durch die Signatur eines Hashwertes nachweist. Analog der Abb. 15 würde man hier die Annahme 2 durch Tatsache 2 ersetzen, was die Authentisierungsunsicherheit reduziert.

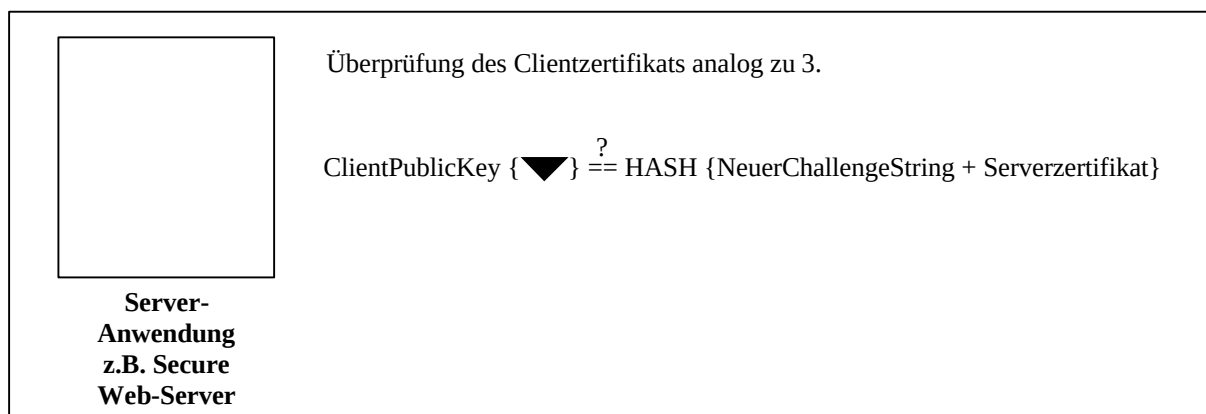


Abb. 19: SSL-Handshake (Client Certificate 3/3)

Fazit 2: Die Schritte 8. bis 11. bewirken die Authentisierung des Clients.

12. Server Finish: Abschließend schickt der Server dem Client eine symmetrisch verschlüsselte Sitzungsidentifikationsnummer (*session identifier, SID*). Der Handshake ist damit abgeschlossen.

2.3.4.3 SSL in Java – die Java Secure Socket Extension (JSSE)

Die beiden vorangegangenen Abschnitte haben wesentliche Eigenschaften des SSL-Protokolls dargestellt. Sie bilden die theoretische Grundlage für das Verständnis, wie die Sicherheitskonzepte Vertraulichkeit und Integrität der Kommunikation sowie Client- und Serverauthentisierung in der Zielplattform umgesetzt werden.

Dieser Abschnitt beschreibt die praktischen Grundlagen der Umsetzung. Wie in Abschnitt 2.2 dargestellt, besteht die Zielplattform serverseitig aus einem hundertprozentig in Java programmierten Application Server. Eine Möglichkeit, das SSL-Protokoll in Java umzusetzen, wird durch die Nutzung der sog. **Java Secure Socket Extension (JSSE)** erreicht. Hierbei handelt es sich um Java-Packages, die eine Implementierung von SSL-Funktionalität in Java bereitstellen. Seit der Version J2SDK v1.4 ist JSSE Bestandteil der Grundentwicklungsumgebung²⁹. Die SSL-Funktionalität ist in drei Packages enthalten (*javax.net*, *javax.net.ssl* und *javax.security.cert*), wobei das Package ***javax.net.ssl*** den Kern bildet. *javax.net* enthält lediglich zwei Klassen, die Abstraktionen des *Socket-Factory*- bzw. des *Server-Socket-Factory-Konzepts* bilden, und *javax.security.cert* dient lediglich der Rückwärtskompatibilität und soll für zukünftige Projekte ausdrücklich nicht verwendet werden³⁰.

Abb. 20 zeigt die wichtigsten Klassen und Interfaces des JSSE und ihren Wirkungszusammenhang. Im Mittelpunkt dieser Darstellung steht die Klasse ***SSLContext***, die zum einen die tatsächlich verwendete SSL- / TLS-Version charakterisiert und zum anderen das Verbindungsglied zwischen Schlüssel- und Vertrauensmaterial sowie Zufallszahlengenerator auf der einen Seite und den Netzwerksockets und ihren Factories auf der anderen Seite bildet. In der praktischen Verwendung ist diese Klasse allerdings nur von Bedeutung, wenn man selbstprogrammierte *KeyManager* bzw. *TrustManager* verwenden möchte. Da in dieser Arbeit die generelle Sicherheitsarchitektur im Mittelpunkt steht und keine optimierte Verwendung des SSL-Protokolls, wird zunächst auf die Verwendung selbstprogrammierter Manager verzichtet. In dieser Arbeit werden zunächst die Standardmanager verwendet, die eine explizite Verwendung der Klasse *SSLContext* überflüssig machen³¹. Wie viele andere Klassen der *Java Security Architecture (JSA)* ist auch die Klasse *SSLContext* als *Engine-Klasse* ausgelegt, wodurch sie zwar einem Verwendungsschema unterliegt, ihre Implementierung aber offen ist. Zusätzliche Implementierungen einer Engine-Klasse können einer Laufzeitumgebung auf einfache Weise durch sog. *Provider* bereitgestellt werden (s. JSA).

Unterhalb der Klasse *SSLContext* sieht man in Abb. 20 zwei Pfade, die jeweils in die Klasse³² ***SSLSession*** münden. Instanzen dieser Klasse repräsentieren Verbindungen zwischen je zwei Teilnehmern. In der Praxis dienen diese Instanzen dazu, wichtige Information über die bestehenden Verbindungen bereitzustellen (z.B. Cipher-Suites, Zertifikate, etc.). Darüber hinaus stellen diese Instanzen für jede dieser Sitzungen ein *Dictionary* bereit, der es ermöglicht, anwendungsspezifische Sitzungsobjekte unter einem Schlüsselnamen abzulegen und wieder zu verwenden. Zugriff auf das betreffende *SSLSession*-Objekt erhält man über den jeweils aktuellen ***SSLSocket***.

²⁹ davor (1.2 und 1.3) war JSSE eine optionale Erweiterung

³⁰ JSSE Reference Guide for J2SDK 1.4

³¹ Verwendung eines default *SSLContext*

³² tatsächlich ist *SSLSession* ein Interface, der Einfachheit halber wird aber nur von Klassen gesprochen

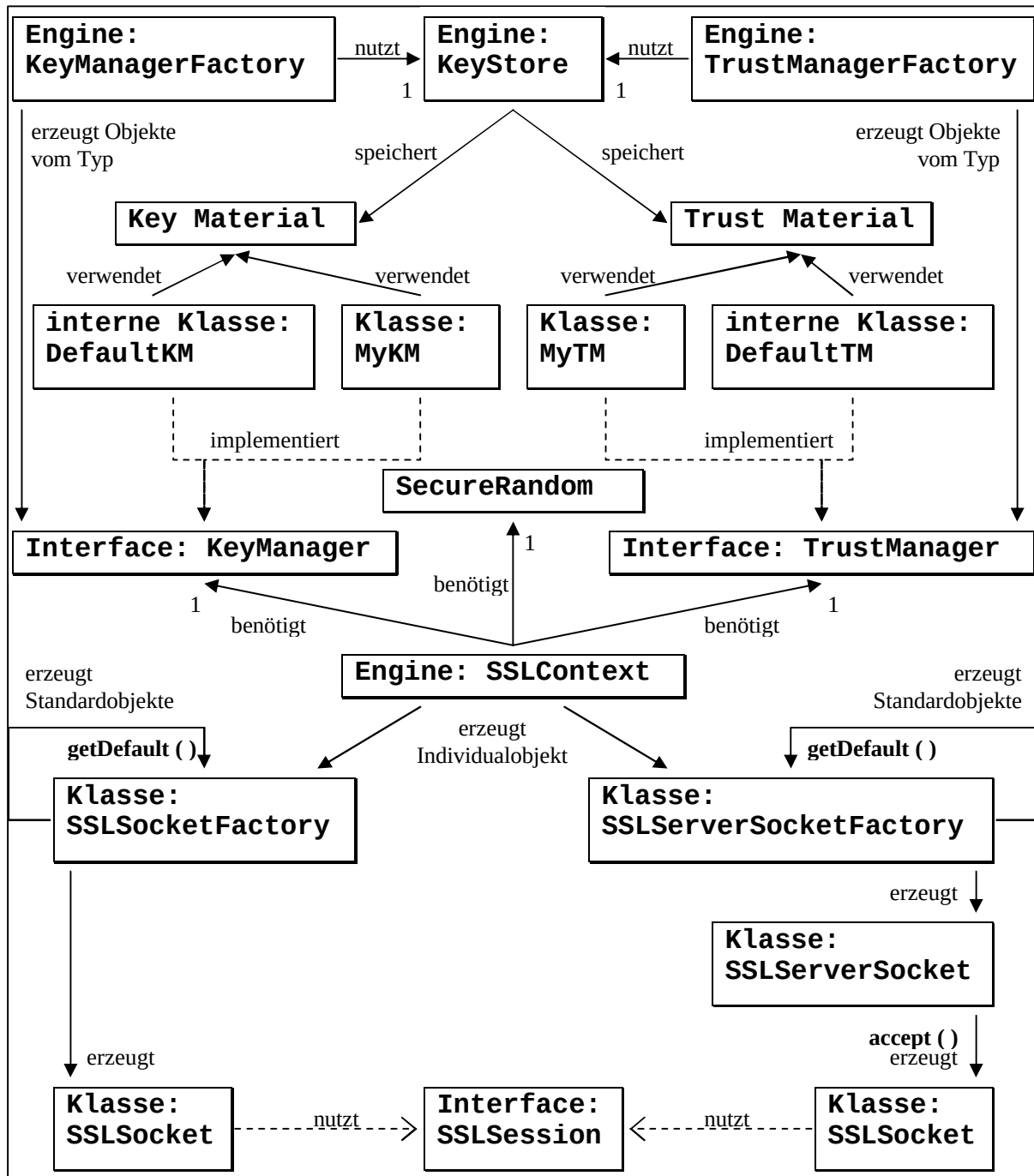


Abb. 20: JSSE Aufbau (Hauptklassen)

Betrachtet man erneut die beiden erwähnten Pfade in Abb. 20, so repräsentiert der linke Pfad den Einsatz von JSSE als Client, während der rechte Pfad den Einsatz als Server repräsentiert. Da für diese Arbeit unter Java lediglich die Serverrolle von Bedeutung ist³³, soll der Einsatz von JSSE ausschließlich in dieser Rolle betrachtet werden.

Geht man von einem individuell konfigurierten **SSLContext** aus, so würde man die betreffende **SSLServerSocketFactory** über eben diese **SSLContext**-Instanz erhalten. Nutzt man hingegen einen Standard-**SSLContext**, so erhält man eine Instanz der **SSLServerSocketFactory** direkt über die Klassenmethode **getDefault()**. Das Ergebnis ist eine individuell konfigurierte oder standardmäßig erzeugte Instanz

³³ clientseitig liegt ein SSL-fähiger Browser vor

der Klasse `SSLServerSocketFactory`, die durch Aufruf der Methode `createServerSocket` eine Instanz der Klasse **`SSLServerSocket`** liefert.

Anhand des `SSLServerSocket`-Objekts, das für einen bestimmten Port konfiguriert ist, kann durch entsprechende Methodenaufrufe eingestellt werden, ob für zukünftige Verbindungen eine Client-Authentisierung erwünscht oder gar notwendig ist. Der Aufruf der Methode `accept` lässt das `SSLServerSocket` auf dem betreffenden Port lauschen, bis eine Anfrage auf dem betreffenden Port eintrifft. Handelt es sich bei der Anfrage um einen neuen Client, für den noch kein `SSLSession`-Objekt existiert, so wird der erforderliche SSL-Handshake durchgeführt (s. 2.3.4.2) und ein neues `SSLSession`-Objekt erzeugt, das für zukünftige Anfragen wieder verwendet wird. Ist der SSL-Handshake erfolgreich (neuer Client) oder liegt bereits ein `SSLSession`-Objekt vor (alter Client), so erhält man als Ergebnis des Methodenaufrufs `accept` eine Instanz von `SSLSocket`, an die auch das `SSLSession`-Objekt geknüpft ist.

Wie in 2.3.4.1 erwähnt und in 2.3.4.2 dargestellt, findet während des SSL-Handshakes üblicherweise eine Server-Authentisierung statt. Zu diesem Zweck benötigt der Server ein SSL-Server-Zertifikat mit zugehörigem privatem Schlüssel. In Abb. 20 ist dies dem Begriff ***Key Material*** untergeordnet. Soll der Server zusätzlich eine Client-Authentisierung während des SSL-Handshakes durchführen, so benötigt er zusätzlich die Angabe eines Zertifikatsspeichers, das die vertrauenswürdigen Stammzertifikate enthält. Dies ist dem Begriff ***Trust Material*** untergeordnet. In der vorliegenden Arbeit werden die betreffenden Zertifikatsspeicher durch Systemeigenschaften (*System-Properties*) festgelegt.

Eine ausführliche Darstellung im Umgang mit der JSSE kann u.a. der Java Online-Dokumentation entnommen werden ([J2Sun02] → JSSE Reference Guide).

2.3.5 SafeGuard Biometrics – ein SmartCard-Produkt der Fa. Utimaco Safeware AG

In diesem Abschnitt wird das eingesetzte SmartCard-Produkt vorgestellt, das im Rahmen dieser Arbeit verwendet wird.

2.3.5.1 Produktauswahl

Bei der Wahl des einzusetzenden kommerziellen Produktes sind die Hauptentscheidungskriterien:

- die Möglichkeit, das Produkt zur webbasierten Authentisierung einsetzen zu können,
- die einfache Integrierbarkeit in das bestehende System zu ermöglichen
- und einen möglichst hohen Sicherheitsgrad zu erreichen.

Die Entscheidung ist aufgrund der von den jeweiligen Produktherstellern veröffentlichten Produktbeschreibungen durchgeführt worden. Keines der Systeme hat vor der Entscheidung für eine Eignungsprüfung zur Verfügung gestanden. In Anbetracht der relativ hohen Kosten für jedes der betrachteten SmartCard-Systeme und der Tatsache, dass die Integration eines solchen Systems nicht die einzige Aufgabe dieser Arbeit ist, wäre eine ausführliche Erprobung verschiedener SmartCard-Produkte auch unangemessen gewesen.

Aufgrund der Tatsache, dass das Produkt ***SmartCard Biometrics*** vom Hersteller Utimaco Safeware AG die obigen Kriterien gemäß der Produktbeschreibung zu erfüllen schien, und in Anbetracht der Tatsache, dass es sich hierbei insbesondere um ein biometriegesichertes Produkt handelt, ist es für die hier erarbeitete

Sicherheitsarchitektur ausgewählt worden. Insbesondere ist die Angabe des Herstellers maßgebend gewesen, dass das Produkt mit „Standardanwendungen von Drittherstellern wie Microsoft Internet Explorer, Netscape oder Outlook“³⁴ zusammenarbeite.

2.3.5.2 Produkteigenschaften

Das Produkt *SmartCard Biometrics* besteht aus einem USB-SmartCard-Lesegerät, in dem ein Fingerabdrucksensor integriert ist, einer SmartCard und einer CD-ROM³⁵. Das Handbuch ist stark marketinglastig und gibt insbesondere Entwicklern, die eine Anwendung aufbauend auf dem Produkt erstellen möchten, nur eingeschränkte Hilfestellung. Dies wird unterstrichen durch den Verweis auf die kostenpflichtige Servicehotline für „Kunden ohne Wartungsvertrag“ (Kap. 1), durch die ausführliche Darstellung der Problemlösungskompetenz weiterer Utimaco Produkte (Kap. 3) sowie durch die schwer verständliche Integration des Systems in gängige Browser, die nur durch ein umfangreiches Ausprobieren zum Ziel führte. Insbesondere das Speichern eines Zertifikats einer webbasierten CA ist mit außerordentlichen Schwierigkeiten behaftet gewesen. Darüber hinaus ist die mit dem Produkt mitgelieferte Software in Bezug auf Zertifikatsmanagement wenig hilfreich.

Abb. 21 gibt einen schematischen Überblick über das SmartCard-System. Die Installation der beiden Treiber erfolgt nach dem Anschließen des SmartCard-Systems. Im Anschluss können die benötigten Anwendungen installiert werden, wobei die *Enrollment Station* und die *SmartCard Provider* für diese Arbeit die wichtigste Rolle spielen. Die *Logon Extensions* ermöglichen, dass sich ein Benutzer mit Hilfe der SmartCard an einem Computer anmelden kann. Die *SmartCard Administration* stellt unter Windows administrative Funktionen für die SmartCard zur Verfügung. Beide zuletzt genannten Anwendungsmodule stehen nur unter Windows 2000 oder Windows XP zur Verfügung.

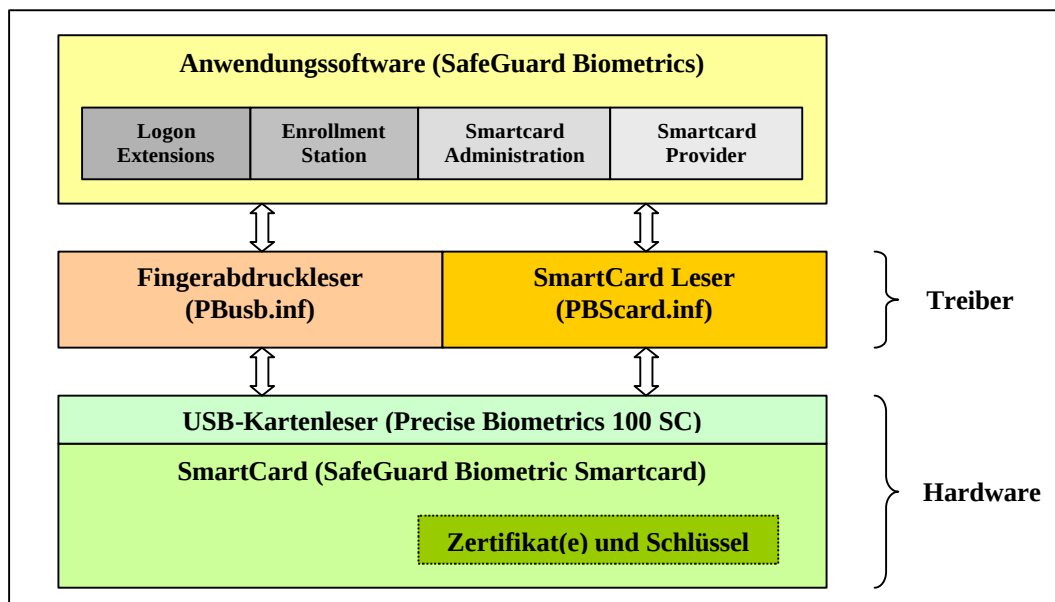


Abb. 21: SmartCard-Architektur

³⁴ www.utimaco.de, Produktbeschreibung: SafeGuard Biometrics

³⁵ Treiber, Anwendungen und elektronisches Handbuch

Die *Enrollment Station* ermöglicht es, die Fingerabdrücke des betreffenden Karteninhabers mit der Karte zu verknüpfen. Der Umgang mit diesem Anwendungsmodul ist sehr ausführlich im Handbuch beschrieben, weshalb hier nicht näher auf Details eingegangen wird.

Die *SmartCard Provider* sind der eigentliche Schlüssel, um das SmartCard-System mit anderen Anwendungen zusammen betreiben zu können. Hierin sind u.a. der PKCS#11-Provider enthalten, der es erlaubt, das System zusammen mit Netscape zu nutzen, oder der Cryptographic Service Provider, der gleiches für den MS Internet Explorer ermöglichen soll. Betrachtet man beispielsweise den PKCS#11-Provider, so dient dieser allgemein dazu, von den Eigenschaften des zugrunde liegenden Kryptogeräts zu abstrahieren, um sie durch allgemeine Kryptofunktionen zu ersetzen. Jede Anwendung, die diese Schnittstelle nutzt, wie beispielsweise Netscape, besitzt also keinerlei Kenntnis über das tatsächlich zugrunde liegende Kryptogerät. Aus Sicht des Browsers existieren damit zunächst einmal keine Unterschiede zwischen den Speicherorten von Zertifikaten (z.B. im Rechner, auf der SmartCard, etc.). Da eine Unterscheidung zur Umsetzung der verschiedenen Authentisierungskonzepte erforderlich ist, kann diese nur aufgrund der Zertifikate und ihrer Ausstellungsrichtlinien geschehen. Besonders wichtig für die einwandfreie Funktionsweise des SmartCard-Systems mit Netscape ist die Registrierung des betreffenden PKCS#11-PlugIns (pkcs201n.dll) mit Hilfe des Geräte-Managers (vgl. 2.3.3.4). Nach diesem Installationsschritt sind ein Generieren eines asymmetrischen Schlüsselpaares auf der Karte und das Ausstellen des korrespondierenden Zertifikates durch eine webbasierte CA möglich.

Für diese Arbeit wurde zu Testzwecken ein Zertifikat der Fa. TC TrustCenter AG (www.trustcenter.de) der Klasse 1 auf der SmartCard installiert. Um die erarbeitete Sicherheitsarchitektur zu testen, ist diese Wahl absolut hinreichend gewesen. Für ein Anwendungsszenario ist natürlich zusätzlich die Frage nach der richtigen Zertifikatsquelle und der richtigen Zertifizierungsrichtlinie zu klären.

3 Vertraulichkeit, Integrität und Authentisierung (V.I.A.)

In diesem Kapitel werden sämtliche Ausführungen zu der Umsetzung der Sicherheitskonzepte *Vertraulichkeit*, *Integrität* und *Authentisierung* erläutert. Die Tatsache, dass drei wichtige Sicherheitskonzepte innerhalb eines einzigen Kapitels abgearbeitet werden, hängt mit der Art ihrer technischen Umsetzung zusammen. In diesem Kapitel wird zunächst auf konzeptioneller Ebene erläutert, wie die genannten Sicherheitskonzepte in der Portalplattform umgesetzt werden. Hierbei wird auf die Grundlagen des Kap. 2 zurückgegriffen. Im Anschluss wird erläutert, welche Überlegungen beim Entwurf maßgeblich gewesen sind, um abschließend auf sämtliche Detaillösungen der Implementierung einzugehen.

3.1 Konzepte zur Umsetzung beim infoAsset Broker

Wie bereits in der Darstellung der Zielplattform (vgl. 2.2) erläutert worden ist, handelt es sich hierbei um eine internetbasierte Client-Server-Architektur, bei der es sich serverseitig um einen javabasierten Application Server und clientseitig um handelsübliche Web Browser handelt. Der wichtigste Standard, um bei einer solchen Architektur die obigen Sicherheitskonzepte umzusetzen, ist die Nutzung des **SSL-Protokolls**. Wie in Abschnitt 2.3.4.2 gezeigt worden ist, setzt dieses Protokoll tatsächlich die genannten Sicherheitskonzepte um, so dass die Umsetzung durch die Wahl dieses Lösungsansatzes einen gangbaren Weg darstellt. Die wichtigste Begründung für die Nutzung dieses Protokolls liegt in der breiten Unterstützung durch marktübliche Web Browser.

Die Umsetzung eines solchen Standardprotokolls würde üblicherweise nicht eine solch umfangreiche Arbeit rechtfertigen, da bereits fertige Module für marktübliche Web Server existieren, die das SSL-Protokoll schlüsselfertig bereitstellen. Die Berechtigung für diese Arbeit liegt in der Tatsache begründet, dass der serverseitige Application Server eben keinen solchen Web Server nutzt. Vielmehr ist der Web Server in den Application Server integriert, so dass die Nutzung des SSL-Protokolls die Auseinandersetzung mit der Architektur der Portalplattform erfordert.

Die Plattform infoAsset Broker ist in Java programmiert, so dass eine Nutzung des SSL-Protokolls eine Umsetzung des Protokolls in Java erforderlich macht. Erfreulicherweise stellt die Programmierumgebung ein Paket zur Umsetzung dieses Protokolls zur Verfügung: die Java Secure Socket Extension (**JSSE**). In der für diese Arbeit verwendeten Version der Programmierumgebung J2SDK 1.4.1. ist diese Erweiterung bereits enthalten. Ihre elementaren Grundlagen sind in Abschnitt 2.3.4.3 erläutert worden. Weitergehende Information zum JSSE können dem *JSSE Reference Guide*, der sich innerhalb der J2SDK Online Dokumentation befindet, entnommen werden.

Zusammenfassend lässt sich sagen, dass das Konzept zur Umsetzung der Sicherheitskonzepte Vertraulichkeit, Integrität und Authentisierung beim infoAsset Broker die Verwendung der in Java bereitgestellten Erweiterung JSSE ist.

3.2 Beschreibung des Entwurfs

Der vorangegangene Abschnitt zeigt, warum die Nutzung der Erweiterung JSSE eine dermaßen wichtige Rolle für diese Arbeit spielt. In diesem Abschnitt werden der Entwurf des Secure Web Servers sowie die zu seiner Entstehung maßgeblichen Überlegungen unter Berücksichtigung der Gegebenheiten bei der JSSE erläutert. An dieser Stelle sei angemerkt, dass im Rahmen dieser Arbeit ein Prototyp eines Secure Web Servers zur Erprobung der JSSE-Technologie erstellt worden ist. Die hieraus resultierenden Erkenntnisse sind in den Entwurf mit eingeflossen.

Bevor allerdings auf den Entwurf eingegangen wird, findet eine Analyse der programmatischen Gegebenheiten beim infoAsset Broker statt, um spätere Erläuterungen verständlicher zu gestalten.

3.2.1 Analyse des bestehenden Web Servers

Die Erweiterung des infoAsset Brokers um einen Secure Web Server impliziert, dass eine ausreichende Kenntnis über die Struktur und das Verhalten des Systems in Bezug auf den vorhandenen Web Server bereits vorliegt. In diesem Abschnitt soll diesbezügliche Information zusammengetragen werden.

Ausgangspunkt der Betrachtung ist die Klasse *WebServer* (*de.infoasset.broker.server.WebServer*), die die Hauptklasse des infoAsset Brokers bildet. Sie enthält die Methode *main*, durch die der Application Server gestartet wird. Zu den ersten Aufgaben dieser Methode zählen das Auslesen der Konfigurationen aus einer Konfigurationsdatei sowie das Erzeugen und Starten eines Web Servers für jede dieser Konfigurationen. Unterschiedliche Web Server können hierbei zu Autorisierungszwecken auf der Basis unterschiedlicher Internetadressenbereiche (Internet, Intranet, etc.) konfiguriert werden. Der Einfachheit halber und weil Autorisierung ebenfalls integraler Bestandteil dieser Arbeit ist, soll in der weiteren Betrachtung davon ausgegangen werden, dass lediglich ein Web Server gestartet wird. Dies ist konsistent mit dem Wunsch, eine feingranulierte Rechtevergabe zu ermöglichen, die nicht auf Adressbereichen beruht. Gleichzeitig stellt dies keine wesentliche Einschränkung dar, da wichtige Abläufe zukünftig ja über den Secure Web Server abgewickelt werden sollen.

Nach dem Starten des Web Servers durch Aufruf der Methode *serve*, wird das erzeugte *WebServer*-Objekt einem neu erzeugten *Thread* übergeben und dieser gestartet. Damit dies möglich ist, implementiert die Klasse *WebServer* das Interface *Runnable*. Gemäß der Konvention löst das Starten des Threads den Aufruf der Methode *run* innerhalb der Klasse *WebServer* aus und zwar in nebenläufiger Weise. Zusammenfassend bedeutet dies, dass alle konfigurierten Web Server die Methode *run* in einem eigenen Thread – also nebenläufig – ausführen.

Innerhalb der Methode *run* wird ein Objekt der Klasse *ServerSocket* generiert und durch Aufruf der Methode *accept* gestartet. Dieses Objekt lauscht somit innerhalb des *WebServer*-Threads auf einem definierten Port auf die Anfragen, die von außen kommen. Währenddessen ruht der Thread. Erfolgt eine Anfrage auf dem vorbestimmten Port, erwacht der Thread und die Methode *accept* liefert als Rückgabewert ein Objekt der Klasse *Socket*. Dieser *Socket* repräsentiert den Kommunikationsendpunkt zum anfragenden Computer und wird als Parameter einem neu erzeugten Objekt der Klasse *HttpConnection* (*de.infoasset.broker.server.HttpConnection*) übergeben, das für die Beantwortung der Anfrage verantwortlich ist. Starten des *ServerSockets* und Beantworten der Anfrage laufen hierbei in einer Endlosschleife, so dass man zusammenfassend

sagen kann, dass jeder *WebServer-Thread* nichts anderes macht, als auf seinem definierten Port zu lauschen und die eintreffende Anfrage durch die Erzeugung eines *HttpConnection*-Objekts zu beantworten. Die Behandlung der Ausnahmesituationen ist hier der Einfachheit halber nicht beschrieben worden.

Die Klasse *HttpConnection* ist eine Spezialisierung der Klasse *Thread*, so dass innerhalb des *WebServer*-Threads jede Anfrage innerhalb eines eben solchen *HttpConnection*-Threads abgearbeitet wird. Hierzu wird am Ende des *HttpConnection*-Konstruktors die Methode *start* aufgerufen, die wiederum dazu führt, dass die Methode *run* aufgerufen wird. Die Methode *run* leitet mit Hilfe des als Parameter empfangenen *Sockets* und der daran gebundenen *Streams* die Beantwortung der Anfrage ein. Im Wesentlichen wird die Anfrage hierbei von der Methode *processRequest* beantwortet, die von der Methode *run* aufgerufen wird. Der *HttpConnection*-Thread stirbt nach dem Beantworten der Anfrage, für die er verantwortlich ist.

Innerhalb der Methode *processRequest* wird die Antwort auf die Anfrage des Clients erstellt und übermittelt. Sie ist verhältnismäßig umfangreich und komplex, so dass hier nicht weiter auf sie eingegangen wird. Dort, wo es bei Detailfragen erforderlich ist, wird auf sie erneut zurückgekommen.

3.2.2 Entwurf: Überlegungen und Kriterien

Neben den Gegebenheiten beim Broker, die im vorangegangenen Abschnitt teilweise beschrieben worden sind, haben beim Entwurf des Secure Web Server vier Kriterien eine tragende Rolle gespielt: Erweiterbarkeit, Kapselung, Konfigurierbarkeit und Wartbarkeit. In welcher Weise jedes dieser Kriterien einen Beitrag zu der Secure Web Server Architektur (**SWS-Architektur**) geleistet hat, soll zunächst kurz erläutert werden.

Das Kriterium **Erweiterbarkeit** ist unter dem Gesichtspunkt zu verstehen, dass mehrere Secure Web Server nebeneinander für einen Application Server koexistieren können sollen. Ein wichtiges Ziel dieser Maßnahme ist es, dass unterschiedliche Authentisierungsstrategien ermöglicht werden und dass nachträglich neue Strategien innerhalb des Systems leicht umgesetzt werden können. So ist beispielsweise denkbar, dass zukünftig neue Sicherheitsgeräte zur Client-Authentisierung genutzt werden sollen, dass eigene Key- oder Trust-Manager implementiert werden sollen, usw. Zum Zwecke der Erweiterbarkeit existiert innerhalb der Java Security Architecture (JSA) das sog. *Engine-Pattern*. Eine Engine-Klasse repräsentiert eine Hülle, die von vielen möglichen Implementierungen ausgefüllt werden kann. Diese Implementierungen werden durch sog. *Provider* bereitgestellt, die leicht in der Laufzeitumgebung registriert werden können³⁶. In Kap. 2 sind diese Engine-Klassen bereits im Kontext der Klasse *KeyStore* und der Klasse *SSLContext* erwähnt worden. Der in dieser Arbeit erstellte Entwurf orientiert sich an dem Engine-Pattern, ohne dass dieser tatsächlich die Umsetzung des Secure Web Servers als Engine-Klasse vorsieht. Vielmehr wird ein Zugriffsmuster ähnlich dem einer Engine-Klasse festgelegt. In dem vorgelegten Entwurf sind die Implementierungen allerdings Spezialisierungen einer abstrakten Secure Web Server Klasse. Auf diese Weise wird nicht nur die mögliche Erweiterbarkeit gesichert, sondern es wird eine wichtige Redesignmöglichkeit eröffnet. Die Architektur kann demnach zu einem späteren Zeitpunkt zugunsten von Engines oder zugunsten des hier gewählten Ansatzes ausgestaltet werden. Für diesen Entwurf bedeutet dies,

³⁶ Details zu Engines und Provider in [Oaks01]

dass für die Klasse *SSLWebServer*, die den Secure Web Server repräsentiert, öffentliche Methodennamen gewählt wurden, die sich an der Namensgebung bei Engines orientiert. Ein Überschreiben bestimmter Methoden ermöglicht es, abweichende Implementierungen durch spezialisierte Klassen bereitzustellen. Wichtig ist hierbei, dass spezialisierte Klassen ein für Engines üblichen Namensattribut enthalten, der sie von anderen Implementierungen unterscheidet.

Das Kriterium **Kapselung** beinhaltet zum einen die Modularisierung des Secure Web Servers und zum anderen die Zugänglichkeit der Funktionalität über wenige, definierte Schnittstellen. Dieses Kriterium wird durch die Tatsache berücksichtigt, dass sich sämtliche, relevanten Klassen des Secure Web Servers mit Ausnahme der erforderlichen Handler innerhalb eines Packages befinden (***de.infoasset.broker.security.ssl***). Zusätzlich existiert eine Klasse namens *SSLWebServerFactory*, die dem umgebenden Programm als Zugriffsschnittstelle dient. Sie enthält eine Methode namens *startSSLServers*, die den Start sämtlicher, konfigurierter Secure Server veranlasst. Somit findet Konfiguration und Verwaltung der Secure Server innerhalb des Packages statt. Ein Benutzer muss lediglich wissen, dass er die Server durch die genannte Methode starten kann.

Das Kriterium **Konfigurierbarkeit** beinhaltet die Möglichkeit, im System vorhandene Secure Server individuell zu starten. Darüber hinaus sollen die für einen einzelnen Secure Server erforderlichen Parameter an einer einzigen Stelle gespeichert werden. Berücksichtigung findet dieses Kriterium durch die Einführung der Klasse *SSLConfiguration*. Objekte dieser Klasse enthalten im Wesentlichen die Konfigurationsdaten einzelner Secure Server. Sie haben damit den Charakter von Konfigurationsobjekten. Wird ein Objekt dieser Klasse instanziiert, so soll mit den Parametern, die es umfasst, auch ein Secure Server gestartet werden. Typische Parameter sind beispielsweise der Port des Servers, die zu verwendenden Key- und Truststores, der Servertyp (vgl. Erweiterbarkeit, Namensattribut des Servers), u.a.m.

Das Kriterium **Wartbarkeit** fordert, dass eine Umgestaltung von Teilen der Architektur und der Komponenten leicht bewerkstelligt werden kann und dass diese Änderungen eine möglichst geringe Pflege des Systems an einer anderen Stelle erfordert. An dieser Stelle ist dieses Kriterium auf die vordefinierten Secure Server zu beziehen. Zur Umsetzung dieses Kriteriums ist in dieser Arbeit der Ansatz gewählt worden, für jeden vordefinierten Secure Server Typ eine eigene Klasse bereitzustellen. Gleichzeitig ist jedem Servertyp ein Handler zugeordnet³⁷ (vgl. Abb. 22), der beim Wechsel von einer ungesicherten zu einer gesicherten Verbindung aufgerufen wird und der die erforderliche Anmeldeprozedur durchführt. Änderungen bei einem Servertyp und dem jeweils korrelierten Handler wirken sich somit lediglich auf einen Anmeldemechanismus aus. Andere Mechanismen bleiben hiervon unberührt.

³⁷ die Zuordnung erfolgt im zugehörigen SSLConfiguration-Objekt und in der handler.txt

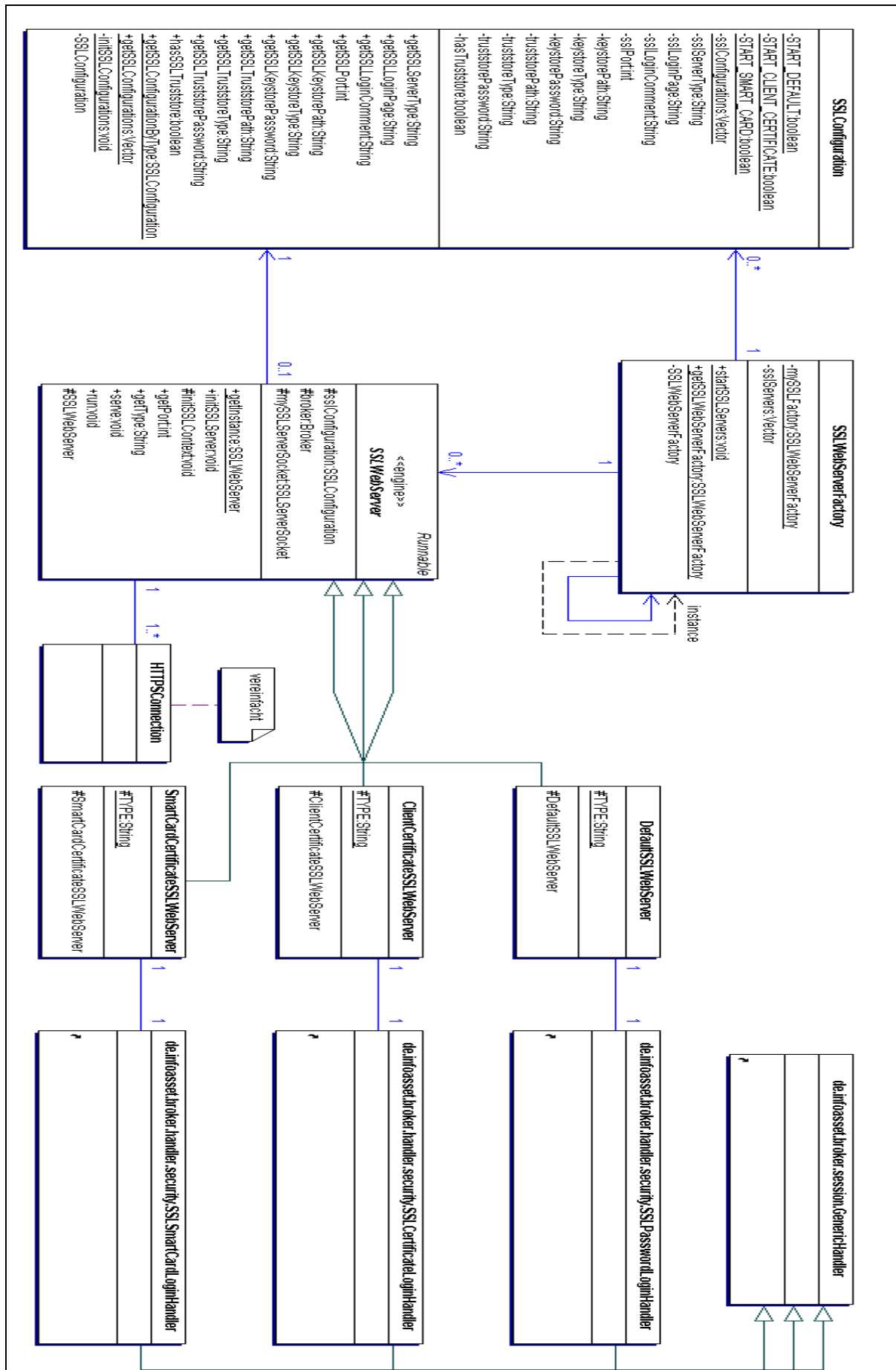


Abb. 22: Entwurf der Secure Web Server Architektur (SWS-Architektur)

3.2.3 Entwurf: Die Gesamtarchitektur

Die im vorangegangenen Abschnitt beschriebenen Kriterien bilden die Grundlage für den in diesem Abschnitt vorgelegten Entwurf. Der Entwurf der SWS-Architektur enthält einige wenige Hauptklassen, die bereits im vorangegangenen Abschnitt z.T. erwähnt worden sind. Diese sollen in diesem Abschnitt näher erläutert werden, um den Aufbau der SWS-Architektur zu verdeutlichen. Alle weiteren Klassen, die im Kontext der SWS-Architektur stehen, sind Hilfsklassen, die sehr spezifische Aufgaben lösen. Auf diese Hilfsklassen wird bei der Erläuterung der Implementierung eingegangen. Sie sind in diesem Entwurf nicht enthalten.

Das Klassendiagramm des Entwurfs (Abb. 22) zeigt die Hauptklassen der implementierten SWS-Architektur. Mit Ausnahme der Handler, die Spezialisierungen der Klasse *GenericHandler* sind, sind alle dargestellten Klassen Bestandteil des Packages *de.infoasset.broker.security.ssl*. Alle Handler der SWS-Architektur sind konform mit der Package-Hierarchie des Brokers im Package *de.infoasset.broker.handler.security* enthalten.

Wie in Abschnitt 3.2.2 (Kapselung) dargestellt, bildet die Klasse **SSLWebServerFactory** die Schnittstelle, über die die SWS-Architektur von außen angestoßen wird. Ihre Hauptaufgabe ist die Verwaltung der konfigurierten SSL-Server, sowie die Funktionalität bereitzustellen, die sämtliche konfigurierten SSL-Server startet. Da diese Aufgaben von einem einzigen Objekt bewerkstelligt werden können, wird diese Klasse gemäß des *Singleton-Patterns* implementiert. Die Methode, die sämtliche konfigurierten SSL-Server startet und die auch das Singleton-Objekt generiert, ist die Klassenmethode **startSSLServers**. Innerhalb dieser Methode werden die vorhandenen Konfigurationen von der verantwortlichen Klasse erfragt und die SSL-Server entsprechend diesen Konfigurationen gestartet.

Verantwortlich für die Konfigurationen ist die Klasse **SSLConfiguration**. Diese Klasse ist so konzipiert, dass Konfigurationen durch *Flags*, die als Konstanten³⁸ implementiert sind, auf einfache Weise ein- oder ausgeschaltet werden können. Auf diese Weise wird ein Testen der SWS-Architektur unter Berücksichtigung unterschiedlicher Konfigurationskonstellationen auf einfache Weise ermöglicht.

Instanzen der Klasse *SSLConfiguration* stellen die Parameter bereit, die für das Initialisieren der SSL-Server benötigt werden. Die Objekte dieser Klasse werden dabei von der Klasse selbst in einer linearen Liste verwaltet. Sämtliche Konfigurationen werden bei einem einmalig stattfindenden Initialisierungsvorgang erstellt und in der genannten Liste abgelegt. Diese Initialisierungsprozedur wird durch die Klassenmethode **initSSLConfigurations** durchgeführt. Innerhalb dieser Initialisierungsmethode werden entsprechend der gesetzten Flags einzelne Konfigurationen erstellt. Die dafür erforderlichen Parameter können an dieser Stelle aus einer Datei, einer Datenbanktabelle, o.ä. ausgelesen werden. Im vorliegenden Entwurf ist allerdings zunächst eine feste Verdrahtung der Parameter innerhalb dieser Methode vorgesehen.

Die von einem Konfigurationsobjekt bereitgestellten Parameter dienen zu Initialisierung der betreffenden SSL-Server. Sie haben im Einzelnen folgende Aufgaben:

- **sslServerType** (String): Dieses Attribut benennt die zu instanziiierende SSL-Server-Implementierung anhand eines charakteristischen Schlüsselwortes. Die

³⁸ finale Klassenattribute

Instanziierung über Schlüsselwörter ist charakteristisch für die Instanziierung einer Engine-Klasse. Aufgrund der vorhandenen SSL-Server-Implementierungen sind hier Werte aus der Menge `{default, client_certificate, smart_card}` erlaubt. Analog einer Engine-Klasse wird für unbekannte Implementierungen die Ausnahme `NoSuchAlgorithmException` ausgelöst.

- **sslLoginPage** (String): Jedem SSL-Server ist ein Handler zugeordnet, über den die Anmeldefunktionalität für den betreffenden SSL-Server implementiert ist. Der hierin befindliche String hat die Syntax „security/sslXXXLogin.htm“, wobei XXX durch einen für den SSL-Server charakteristischen Namen zu ersetzen ist. Dieser String ist außerordentlich wichtig, da er zur Generierung einer dynamisch erzeugten Verknüpfung dient, die für den Aufruf des zugeordneten Handlers sorgt (vgl. *de.infoasset.broker.handler.myportal.WelcomeHandler*, *ssl-Servers-ListTemplate*).
- **sslLoginComment** (String): Dieser Parameter spielt eine untergeordnete Rolle für die Funktionsweise der SWS-Architektur. Er enthält eine kurze Informationszeile, die zur Erläuterung der dynamischen Verknüpfung verwendet wird. Durch sie erlangt der Benutzer Kenntnis über die Art der Anmeldung.
- **sslPort** (int): Dies ist der Port, den der betreffende SSL-Server öffnet. Der hier eingetragene Port wird ebenfalls bei der dynamischen Erzeugung der Verknüpfung verwendet.
- **keystorePath** (String): Dies ist der vollständige Name der Keystore-Datei, die der betreffende SSL-Server verwenden soll. Gemäß den JSSE-Vorgaben bei der Nutzung des Standard-SSLContextes soll diese Datei nur ein Zertifikat enthalten. Dieses Zertifikat und der korrelierte private Schlüssel dienen zur Server-Authentisierung. Dementsprechend handelt es sich um ein SSL-Server-Zertifikat. Es sollte von einer bekannten CA ausgestellt worden sein, deren Stammzertifikat(e) in gängigen Web Browsern vorinstalliert ist bzw. sind, beispielsweise TC TrustCenter, GlobalSign, VeriSign, etc.
- **keystoreType** (String): Dies bezeichnet die Keystore-Implementierung, die zur Generierung der Keystore-Datei verwendet wurde. Standardmäßig stehen hier drei Varianten zur Verfügung: *JKS*, *JCEKS* oder *PKCS12*.
- **keystorePassword** (String): Dieser Parameter enthält das Passwort, das zum Schutz der Integrität der Keystore-Datei verwendet wird. Dieses Passwort schützt gleichzeitig auch die Verwendung des in dieser Datei enthaltenen privaten Schlüssels. Gemäß der JSSE-Konvention für die Nutzung des Standard-SSLContextes müssen diese beiden Passwörter identisch sein³⁹. Bei Keystores, die zu einem anderen Zweck verwendet werden, muss dies natürlich nicht so sein.
- **truststorePath** (String): Dies ist der vollständige Name der Truststore-Datei, die der betreffende SSL-Server verwenden soll. Eine Truststore-Datei ist die Keystore-Datei, die ausschließlich die vertrauenswürdigen Stammzertifikate enthält, die der betreffende Server verwenden soll. Ihre Einträge dienen somit zur Client-Authentisierung. Aufgrund der Tatsache, dass der Client Zertifikate mit unterschiedlichen Richtlinien verwenden kann, kann mit Hilfe der im Truststore installierten Zertifikate eine Unterscheidung der Authentisierungsgüte vorgenommen werden. Für den Fall, dass der SSL-Server keine Client-Authentisierung via SSL-Handshake erfordert, kann dieser Parameter auch *null* sein.

³⁹ vgl. [Oaks01], SSL and HTTPS, Keystores and Truststores

- **truststoreType** (String): siehe *keystoreType* (angewandt auf Truststore-Datei).
- **truststorePassword** (String): siehe *keystorePassword* (angewandt auf Truststore-Datei). Da eine Truststore-Datei keine privaten Schlüssel enthält, die zu schützen wären, werden hierfür keine Passwörter benötigt, so dass hier die Frage nach der Gleichheit entfällt.
- **hasTruststore** (boolean): Dieser Wert ist *true*, wenn beim SSL-Handshake eine Client-Authentisierung durchgeführt werden soll, zu der ein Truststore benötigt wird. Anderenfalls ist der Wert *false*.

Die abstrakte Klasse **SSLWebServer**, die in ihren Schnittstellen einer Engine-Klasse gleicht, verkörpert, wie der Name erahnen lässt, das Herzstück der SWS-Architektur. Obwohl sie als Objekt nicht instanzierbar ist, enthält sie die vollständige Funktionalität eines SSL-Servers. Die Implementierungen von instanzierbaren SSL-Servern erfolgen durch Spezialisierung dieser abstrakten Klasse. Das einzige, das die spezialisierten Klassen implementieren müssen, ist eine Konstante mit dem charakteristischen Implementierungsnamen und ein parameterloser Konstruktor. Selbstverständlich können diese spezialisierten Klassen, die Methoden von *SSLWebServer* überschreiben, die vom Standardverhalten abweichen sollen. Die in dieser Arbeit erstellten SSL-Server-Implementierungen behalten alle die Funktionalität ihrer generalisierten Klasse *SSLWebServer* bei. Die tatsächliche Unterscheidung in ihrem Verhalten resultiert somit lediglich aus einer unterschiedlichen Konfiguration. Eine Umgestaltung dieser Umsetzung des Erweiterbarkeitsprinzips ist somit auf einfache Weise möglich, da die bisher vorhandenen spezialisierten Klassen gegenwärtig weitestgehend zur Typisierung dienen. Aus Abb. 22 kann man entnehmen, dass jedem spezialisierten SSL-Server ein Handler zugeordnet ist. Dieser übernimmt die Anmeldeprozedur für den spezifischen SSL-Server. Auf Einzelheiten der Anmeldeprozedur wird im Abschnitt, der die Implementierung betrifft, eingegangen.

Die Klasse *SSLWebServer* implementiert das Interface *Runnable*, damit jeder SSL-Server innerhalb seines eigenen Threads läuft. Die Implementierung dieses Interfaces erfordert die Implementierung der Methode *run*. Innerhalb dieser Methode läuft eine Endlosschleife, die eingehende Anfragen empfängt und diese bearbeitet. Das Prinzip ist somit das gleiche wie bei den Servern ohne SSL. Der wesentliche Unterschied liegt hierbei in den verwendeten *Sockets* und *ServerSockets*. SSL-Server verwenden *SSLSockets* und *SSLServerSockets* der JSSE-Architektur (*javax.net.ssl*).

Ein SSL-Server enthält zwei wichtige Attribute: ein Konfigurationsobjekt (Instanz von *SSLConfiguration*) und ein Objekt der Klasse *SSLServerSocket*. Daneben hält der SSL-Server aus Kompatibilitätsgründen mit der bisherigen Architektur das Broker-Objekt⁴⁰. All diese Attribute werden beim Initialisieren des SSL-Servers zugeordnet.

Die Verwendung eines SSL-Servers erfolgt gemäß dem *Engine-Pattern* über die Klassenmethode ***getInstance (String type)***. Diese liefert die durch den Typ spezifizierte SSL-Server-Implementierung. Ist keine Implementierung eines solchen Namens bekannt, so wird die bei Engines für diesen Fall übliche Ausnahme *NoSuchAlgorithmException* ausgelöst. Bei der in dieser Arbeit vorgelegten Implementierung findet ein Vergleich des übergebenen Typnamens mit denen der vorhandenen Implementierungen statt. Erweitert man die SWS-Architektur um weitere SSL-Server-Implementierungen, so ist dementsprechend diese

⁴⁰ *Broker* ist eine eigene Klasse im *infoAsset Broker*

Klassenmethode anzupassen⁴¹, so dass die neuen Implementierungen als existent erkannt werden.

Wie es oftmals bei Engine-Objekten üblich ist, müssen diese nach dem Instanzieren initialisiert werden. Hierfür existiert kein standardisierter Methodenname wie es für das Konstruieren des Objektes der Fall ist. Für die Initialisierung der SSL-Server geschieht dies durch Aufruf der Methode **initSSLServers**. Die wichtigste Aufgabe dieser Methode ist das Erstellen eines richtig konfigurierten *SSLServerSockets*. Diese Aufgabe erfolgt in zwei Schritten: Initialisieren des *SSLContextes* und Kreieren des *SSLServerSockets*. Der *SSLServerSocket* wird dann als Teil des SSL-Servers gehalten und kann von diesem beliebig oft genutzt werden. Außer dem generierten *SSLServerSocket* werden die der Methode übergebenen Parameter, bestehend aus Konfigurationsobjekt und Broker-Objekt, Attributen des SSL-Servers zugewiesen.

Die vorliegende Implementierung führt eine Initialisierung des *SSLContextes* durch Aufruf der Hilfsmethode **initSSLContext** durch, in der eine Initialisierung des *Standard-SSLContextes* mit Hilfe von Systemeigenschaften (System-Properties) stattfindet. Anschließend wird ein *Standard-SSLServerSocket* generiert⁴². Dies ist die einfachste Möglichkeit, ein *SSLServerSocket* zu erstellen⁴³.

Nachdem der SSL-Server initialisiert ist, kann dieser gestartet werden. Analog der Server-Implementierung ohne SSL existiert hierzu die Methode **serve**, die im Wesentlichen die jeweilige *SSLWebServer-Instanz* in einem eigenen Thread startet. Innerhalb dieses Threads läuft eine Endlosschleife, in der das *SSLServerSocket* auf dem betreffenden Port auf eingehende Anfragen lauscht. Erhält der *SSLServerSocket* eine Anfrage, so generiert er einen *SSLSocket*.

Da in der Java-Klassenhierarchie *SSLSocket* eine Spezialisierung der Klasse *Socket* ist, sollte die in 3.2.1 erwähnte Klasse *HttpConnection* die Anfrage auch mit einem speziellerem Socket beantworten können. Konkret scheitert dies allerdings an einem anderen Parameter, der von dem Konstruktor der Klasse *HttpConnection* erwartet wird: Dieser erwartet nämlich zusätzlich das *WebServer-Objekt* als Parameter. Möchte man somit die Klasse *HttpConnection* für die Beantwortung der Anfrage wieder verwenden, so gibt es zwei Optionen: die Klasse *SSLWebServer* als Unterklasse der Klasse *WebServer* auszuführen und der Anpassung der Klasse *HttpConnection*. Die erste Variante würde bedeuten, dass man keine vollständige Trennung der SWS-Architektur vom restlichen Code mehr hätte und dass man sich dadurch möglicherweise weitere Probleme einhandeln würde. Die zweite Variante beinhaltet, eine Klasse mit einem sehr komplexen Verhalten teilweise umzuschreiben und erprobten, funktionsfähigen Code auf diese Weise zu gefährden. Da offenbar beide Lösungen u.U. neue Probleme schaffen, wird ein Lösungsansatz gewählt, der sich an der zweiten Variante orientiert, ohne diese tatsächlich umzusetzen. Der Lösungsansatz beinhaltet die Wiederverwendung des Codes der Klasse *HttpConnection*, ohne die Klasse wieder zu verwenden. Zu diesem Zweck wird der Code dieser Klasse in eine neu geschaffene Klasse **HTTPSConnection** übertragen und dort entsprechend den Bedürfnissen der SWS-Architektur angepasst. U.a. erwartet die Klasse nunmehr ein *SSLSocket*, so dass auch der Name *HTTPSConnection* gerechtfertigt ist.

⁴¹ dies ist bei üblichen Engines nicht nötig

⁴² vgl. 2.3.4.3

⁴³ Abweichend von dieser Strategie wäre es möglich, das *SSLContext-Objekt* eigenhändig zu instanzieren und zu initialisieren. Aufgrund des höheren Aufwandes ist dies nur sinnvoll, wenn ein vom Standard abweichendes Verhalten erreicht werden möchte. Um an dieser Stelle ein konkretes Beispiel zu nennen, wäre dies beispielsweise sinnvoll, wenn man CRLs implementieren möchte.

3.3 Details der Implementierung

Im vorangegangenen Abschnitt ist die SWS-Architektur ausführlich erläutert worden, ohne jedoch auf die Lösung sehr spezieller Probleme in diesem Kontext einzugehen. Aufgrund dessen ist u.a. bisher die Beschreibung der Funktionalität von vier Klassen des Packages *de.infoasset.broker.security.ssl*, in dem sich die Klassen der SWS-Architektur befinden, ausgeblieben. Die speziellen Problemlösungen und die Aufgaben der verbleibenden Klassen sollen in diesem Abschnitt näher beschrieben werden.

3.3.1 Dynamische Link-Erzeugung und JavaScript

In Abschnitt 3.2 ist auf die serverseitige Architektur der SSL-Server, auf ihre Konfiguration, ihre Generierung und auf die Weise, wie Anfragen beantwortet werden, ausführlich eingegangen worden. Die Frage, wie von einer ungesicherten zu einer gesicherten (SSL-)Verbindung gewechselt wird, ist zunächst ausgeklammert worden. Die Beantwortung dieser Frage erscheint zwar trivial, ist in der programmiertechnischen Praxis allerdings eine erhebliche Hürde.

Um die Problematik zu verstehen, sind das Erkennen clientseitiger Eigenschaften und deren Zusammenwirken erforderlich. Die erste Eigenschaft betrifft das clientseitige Layout, welches mit Hilfe von *Frames* strukturiert ist. Die Nutzung von *Frames* tritt häufig auf. Der Kerngedanke hierbei ist, dass häufig nur einzelne Ausschnitte der Darstellung geändert werden sollen, so dass das Gesamtbild der Website einen einheitlichen Eindruck vermittelt. Der am häufigsten wechselnde *Frame* ist hierbei der sog. *Content-Frame*, in dem die tatsächlich abgerufene Nutzinformation dargestellt wird. Hierbei wird die neu geladene HTML-Seite innerhalb desselben *Frames* an Stelle der bisherigen HTML-Seite dargestellt. Die HTML-Seiten der umliegenden *Frames* müssen dabei nicht neu geladen werden. Bei der Portalplattform funktioniert dies ähnlich: Der Aufruf einer Verknüpfung verursacht häufig nur die Aktualisierung des *Content-Frames*. Um von einer ungesicherten auf eine gesicherte Verbindung zu wechseln, ist der dargestellte Sachverhalt insofern problematisch, als das man sämtliche *Frames* inkl. *Frameset* komplett neu laden muss und hierbei auch den Wechsel der Protokollangabe (https statt http) und des Ports (z.B. 443 statt 80) durchzuführen hat.

Die Lösung dieses Problems erscheint zunächst einfach. Anstatt für den Wechsel der Verbindung eine relative Verknüpfung zu verwenden, benutzt man eine absolute Verknüpfung, die das neu zu verwendende Protokoll und den neu zu verwendenden Port enthält. Zusätzlich gibt man als Ziel der Verknüpfung das oberste *Frame* des Browsers an, so dass das gesamte *Frameset* neu geladen wird. Leider werden in diesem Kontext allerdings weitere Probleme identifiziert.

Der Grund hierfür ist die Tatsache, dass die Verknüpfungen in der Portalplattform keine Hyperlinks sind, die eine URL enthalten, sondern clientseitig eingebetteten JavaScript-Code aufrufen. Ein relativer Hyperlink hätte beispielsweise die Gestalt

`href = "/kategorie/aktion.htm".`

Stattdessen haben die Verknüpfungen häufig die Gestalt

`href = "javascript: parent.session.show ('kategorie/aktion.htm')".`

Die Übergabe einer absoluten Verknüpfungsadresse über bestehenden JavaScript-Code ist somit unmöglich. Entweder setzt man also eine absolute Verknüpfung ein, bricht mit der Konvention der Portalplattform und nimmt dafür möglicherweise weitere Probleme in Kauf, oder man erstellt neuen JavaScript-Code, der die Konvention der Plattform und die spezielle Problematik berücksichtigt. Es dürfte offensichtlich sein, dass die zweite Variante unter den gegebenen Umständen wesentlich eleganter ist.

Zu diesem Zweck wird die JavaScript-Funktion *SSLLogin* (*sslTarget*, *sslPort*) eingeführt⁴⁴, dessen Code hier aufgeführt ist:

```
function SSLLogin (sslTarget, sslPort) {  
    var URL = '';  
  
    alert ('SSL-Verbindung wird über Port ' + sslPort + ' eingeleitet!');  
  
    URL = 'https://' + top.document.location.hostname + ':' + sslPort;  
    URL = URL + '/de/main.htm?t=';  
    URL = URL + sslTarget;  
  
    top.document.location.replace (URL);  
}
```

Diese Funktion soll näher erläutert werden, da sie eine Schlüsselrolle bei dem Wechsel von einer ungesicherten zu einer gesicherten Verbindung spielt. Zunächst wird ein URL-String mit einem leeren String initialisiert und der Benutzer auf den Wechsel zu einer gesicherten Verbindung über ein Meldefenster im Browser hingewiesen. Dem URL-String werden die Protokollbezeichnung „https“, der Domänenname, der aus der JavaScript-DOM⁴⁵ entnommen wird, und der Port, über den die gesicherte Verbindung erfolgen soll, zugewiesen. Durch die Protokollbezeichnung „https“ erreicht man die Verwendung des SSL-Protokolls. Das Auslesen des Hostnamens ermöglicht, dass der Verbindungswechsel flexibel auf unterschiedliche durch die Portalplattform verwendete Domänen reagiert. Die Angabe des Ports ermöglicht schließlich den Zugriff auf den richtigen SSL-Server. Der Port ist hierbei ein Parameter der Funktion, da diese ja für unterschiedlich konfigurierte SSL-Server verwendet werden soll.

Dem URL-String wird im Anschluss der String `/de/main.htm?t=` und ein sog. *Target*, welches die Funktion ebenfalls als Parameter erhält, angehängt. Hierdurch wird das komplette *Frameset* neu geladen. Im *Content-Frame* wird hierbei das *Target* geladen, so dass der Parameter *sslTarget* die Syntax „kategorie/aktion.htm“ haben sollte.

Abschließend wird die JavaScript-Funktion *replace* des Objekts *location* ausgeführt. Diese lädt die Seite mit der übergebenen URL, die folgende Gestalt hat:

<https://<hostname>:<sslPort>/de/main.htm?t=<sslTarget>>

Hierbei werden die Parameter *sslPort* und *sslTarget* durch diejenigen Werte ersetzt, die die Funktion erhält. Der Parameter *hostname* repräsentiert die Domäne der unsicheren Verbindung, da ja SSL-Server und WebServer ohne SSL auf dem gleichen Rechner laufen, und somit der Hostname nicht verändert werden muss. Die Funktion *replace* überschreibt gleichzeitig auch den History-Eintrag im Browser.

⁴⁴ .../Broker/projects/broker/templates/html/de/home/default.htm

⁴⁵ DOM: Domain Object Model, s. JavaScript Tutorial (<http://js-tut.aardon.de/js-tut/>), Anhang A

Der Aufruf der JavaScript-Funktion *SSLLogin* bewirkt somit, dass serverseitig der SSL-Server mit der entsprechenden Portnummer in den Verbindungsaufbau involviert wird. Dieser führt zunächst einen SSL-Handshake entsprechend seiner Konfiguration durch. Bei einem erfolgreichen Handshake kommt eine Verbindung zustande, bei einem erfolglosen Handshake zeigt der Browser des Clients hingegen eine Fehlermeldung an, die außer einem Fehlercode keine weitere Information über die Art des Fehlers liefert. Ist die Verbindung zustande gekommen, so wird das gesamte *Frameset* über die sichere SSL-Verbindung neu geladen, wobei im *Content-Frame* die Seite, die durch den Parameter *sslTarget* definiert ist, geladen wird.

Angemerkt sei noch, dass bei diesem Vorgang eine neue Sitzung mit einer neuen *sessionId* eingeleitet wird. Dies ist in so fern unproblematisch, als dass der Client vor dem Durchführen des sicheren Verbindungsaufbaus nicht angemeldet gewesen ist und somit keine wichtigen Aktivitäten durchgeführt haben kann.

Der Programmcode der Funktion *SSLLogin* befindet sich zunächst im Template ***de/home/default.htm***, kann allerdings später in eine entsprechende JavaScript-Datei (z.B. *content.js*) ausgelagert werden. Die HTML-Seite, die aus diesem Template generiert wird, wird beim Aufrufen des Portals zu Beginn angezeigt. Aufgrund dessen bietet sich dieses Template für den Einbau der dynamischen Verknüpfungen an. Der Handler, der dieses Template instanziiert, ist ***de.infoasset.broker.handler.myportal.WelcomeHandler***. Das Template enthält somit die Platzhalter, die in Abhängigkeit der Programmlogik des Handlers die entsprechenden Verknüpfungen für den Aufbau einer sicheren Verbindung generieren. Da dieser Sachverhalt etwas komplex ist, wird dieser Template-Code hier aufgeführt, um ihn zu erläutern:

`[$isNotLoggedIn$` (1)

```
<tr><td><b>Anmeldung</b></td><td></td></tr>

[$sslServers $s
  <tr align="left" valign="baseline">
    <td> [<a href="javascript:
      SSLLogin ('$s.sslLoginPage$', $s.sslPort$)">Anmelden</a>] (3)
    </td>
    <td>$s.sslLoginComment$</td>
  </tr>
]$sslServers[$
  <tr align="left" valign="baseline">
    <td></td>
    <td><b>Zur Zeit ist leider kein SSL-Server für die Anmeldung
      konfiguriert!</b></td>
  </tr>
$sslServers]$
```

`[$isNotLoggedIn[$` (2)

```
<tr><td><b>Abmeldung</b></td><td></td></tr>

<tr align="left" valign="baseline">
  <td> [<a href="javascript:
    parent.session.show ('security/sslLogoff.htm')">Abmelden</a>]</td>
  <td>Klicken Sie hier, um sich abzumelden.</td>
</tr>
```

`$isNotLoggedIn]$`

Zunächst findet eine Unterscheidung statt, ob der jeweilige Client bereits angemeldet ist oder nicht. Im Template äußert sich dies durch den Platzhalter *isNotLoggedIn*. Für den Fall, dass der Client nicht angemeldet ist **(1)**, wird der erste Teil des Templates berücksichtigt. Im zweiten Teil des Templates wird für den Fall, dass der Client doch angemeldet ist **(2)**, eine Verknüpfung erzeugt, die eine Abmeldeprozedur einleitet.

Innerhalb des ersten Template-Teils befindet sich der Platzhalter *sslServers*, der im Handler von einem sog. *ListTemplate* bearbeitet wird. Auf diese Weise wird dynamisch für jeden konfigurierten SSL-Server eine eigene Verknüpfung erzeugt, die wiederum eine eigene Anmeldeprozedur einleitet. Wie aus dem Template-Code ersichtlich ist, ist das Ziel der Verknüpfung die bereits dargestellte Funktion *SSLLogin*. Für den Fall, dass keine SSL-Server konfiguriert sind, wird dies dem Benutzer durch einfache Ausgabe mitgeteilt.

Wie unschwer zu erkennen sein dürfte, bildet der Listenplatzhalter *sslLoginPage* **(3)** innerhalb der Funktion *SSLLogin (sslTarget, ...)* das Ziel, welches innerhalb des *Content-Frames* geladen wird. Der Listenplatzhalter *sslLoginPage* spezifiziert somit den für die Anmeldeprozedur verwendeten Handler. Diese relative Adressierung wird bei der Instanziierung der Verknüpfung dem jeweiligen Konfigurationsobjekt eines SSL-Servers entnommen.

Die Instanziierung des Templates erfolgt wie erwähnt durch den *WelcomeHandler*. Hier ist Programmcode für die Bearbeitung des *ConditionalTemplates isNotLoggedIn* und für das *ListTemplate sslServers* hinzugefügt worden. Innerhalb des *ListTemplates* wird über die vorhandenen Konfigurationsobjekte für SSL-Server iteriert und hieraus die drei benötigten Parameter entnommen, so dass tatsächlich nur Verknüpfungen für existente Konfigurationen generiert werden. Der hierfür erforderliche Programmcode folgt der Konvention der Portalplattform bei der Instanziierung von *ConditionalTemplates* und *ListTemplates*, so dass hier keine Besonderheiten vorhanden sind. Aus diesem Grund wird dieser Code nicht weiter erläutert.

3.3.2 Die Anmeldefunktionalität

Im vorangegangenen Abschnitt ist erläutert worden, wie clientseitig der Wechsel zu einer gesicherten Verbindung eingeleitet wird. In diesem Abschnitt wird der Schwerpunkt auf die in diesem Kontext stehenden serverseitigen Abläufe gesetzt.

Macht der Client die Anforderung einer sicheren Verbindung durch Einbeziehung eines *https-Requests*, so ist zunächst der SSL-Handshake in einer Weise zu erfüllen, wie dieser durch die Konfiguration des SSL-Servers bestimmt ist (vgl. 2.3.4.3). Für den Client stellt dies insbesondere dann eine Hürde dar, wenn die Konfiguration Client-Authentisierung vorsieht. Kommt eine erforderliche Client-Authentisierung während des SSL-Handshakes nicht zustande, so wird die gesamte Verbindung geblockt, ohne dass irgendwelche Handler involviert wären. Auf diese Vorgänge hat man programmatisch nur durch die Konfiguration der *SSLServerSockets* Einfluss.

Ist nach einem Verbindungsversuch durch den Client der SSL-Handshake erfolgreich abgelaufen, so wird zunächst derjenige Handler in die Anmeldeprozedur involviert, der durch den *Target*-Parameter in der URL und durch den entsprechenden Eintrag in der *handler.txt* festgelegt ist. Hierbei existiert für jeden vorkonfigurierten SSL-Server je ein Handler, der die entsprechende Anmeldeprozedur durchführt. Für die drei vorkonfigurierten SSL-Server existieren somit drei Anmelde-Handler (vgl. Abb. 22). Diese Handler befinden sich in Übereinstimmung mit der Konvention beim

Broker im Package **de.infoasset.broker.handler.security**. Die betreffenden Handler heißen **SSLPasswordLoginHandler**, **SSLCertificateLoginHandler** und **SSLSmartCardLoginHandler**. Jeder dieser Handler führt am Ende der Anmeldeprozedur die Instanziierung eines Templates durch, die jeweils nichts anderes tun, als den clientseitigen Benutzer über die erfolgreiche Anmeldung zu informieren und eine Umleitung auf die Seite *home/default.htm* einzuleiten. Da zu diesem Zeitpunkt bereits das gesamte *Frameset* über die gesicherte Verbindung geladen worden ist, handelt es sich hierbei lediglich um den Austausch des *Content-Frames*. Nach dem Laden der Seite *home/default.htm* sind somit sämtliche *Frames* über die gewählte gesicherte Verbindung geladen, und der Benutzer hat sich auf die von ihm gewünschte Weise gegenüber dem System authentisiert.

Im Einzelnen sind die *SSLxxxLoginHandler* sehr ähnlich. Weil serverseitig nur das Zertifikat aber nicht dessen Herkunft, d.h. die clientseitig verwendete Sicherheitseinrichtung des Browsers, geprüft werden kann, sind die Handler *SSLCertificateLoginHandler* und *SSLSmartCardLoginHandler* sogar fast identisch. Eine Trennung der Programmlogik ist insofern trotzdem gerechtfertigt, als dass in dieser Arbeit keine endgültige Entscheidung bzgl. des Unterscheidungskriteriums der Zertifikate getroffen worden ist. Die Unterscheidung beruht zunächst allein auf dem vom jeweiligen SSL-Server verwendeten *Truststore* (siehe auch Exkurs).

Grob lässt sich der Ablauf in den *SSLxxxLoginHandler* in vier Abschnitte gliedern: Verifizierung der Anmeldeannahmen, die Gewinnung der Anmeldedaten, die eigentliche Anmeldung und die Instanziierung des Umleitungstemplates. Die **Verifizierung der Anmeldeannahmen** soll sicherstellen, dass der betreffende Handler tatsächlich im Anschluss an einen korrelierten SSL-Handshake aufgerufen worden ist. Hiermit werden somit potentielle Eindringungsversuche abgewehrt. Im Fehlerfall wird die Anmeldeprozedur abgebrochen und die Programmlogik auf einen Fehler-Handler (*SSLFailureRedirectHandler*) umgeleitet.

Zunächst wird geprüft, ob der betreffende Handler überhaupt über eine gesicherte Verbindung aufgerufen wird. Dies ist der Fall, wenn ein *SSLSocket* für die Verbindung verwendet wird. Anschließend wird der tatsächlich verwendete Port mit dem Port verglichen, auf dem die betreffende Anmeldeprozedur erwartet wird. Auf diese Weise wird verhindert, dass ein Handler durch die Nutzung eines nicht vorgesehenen *SSLSockets* aufgerufen wird. Sind diese beiden Kriterien erfüllt, so kann davon ausgegangen werden, dass der von dem betreffenden Anmelde-Handler erwartete SSL-Handshake erfolgreich durchgeführt worden ist.

Die **Gewinnung der Anmeldedaten** unterscheidet sich bei den verschiedenen Handlern. Im Falle einer Passwort-Authentisierung⁴⁶ wird das Objekt der Klasse **SSLPasswordCredentials** geholt, das zu der betreffenden Session gehört. Hierbei handelt es sich um ein Hilfsobjekt der SWS-Architektur, das dazu dient, die Credentials, die aus User-Login und Passwort bestehen, für die Anmeldeprozedur im Handler zwischenzuspeichern. Das Objekt wird innerhalb des Threads *HTTPSConnection* erzeugt und innerhalb des *Dictionaries* des *SSLSession*⁴⁷-Objekts zwischengespeichert. Die Gewinnung der erforderlichen Initialisierungsdaten für die Erzeugung des Objekts erfolgt durch http-Authentisierung (vgl. 2.3.1.2). Nach erfolgreicher Anmeldung wird das Objekt aus der *SSLSession* entfernt.

⁴⁶ d.h. keine Client-Authentisierung via SSL-Handshake

⁴⁷ nicht zu verwechseln mit dem Session-Objekt des Brokers

Bei der zertifikatsbasierten Authentisierung wird ein Hilfsobjekt der Klasse **SSLInformationInvestigator**, die ebenfalls Bestandteil der SWS-Architektur ist, einbezogen. Objekte dieser Klasse stellen eine Schnittstelle zur Verfügung, die benötigte Information über die SSL-Verbindung liefert. Komplexe Algorithmen, die beispielsweise zur Extraktion von Anmeldedaten aus einem Zertifikat dienen, können so durch Methoden dieses Objekts gekapselt werden. In der zu dieser Arbeit vorgelegten Implementierung liefert das Objekt konkret die in einem Zertifikat eingetragene E-Mail-Adresse des Zertifikatsinhabers, die in der Architektur der Portalplattform das einzigartige (*unique*) User-Login repräsentiert.

Sind bei der Gewinnung der Anmeldedaten die erforderlichen Credentials nicht zu erlangen oder liefern sie unerwartete Ergebnisse, so wird ebenfalls auf den bereits erwähnten Fehler-Handler umgeleitet. Sind die gewünschten Anmeldedaten allerdings beigebracht, so findet die **eigentliche Anmeldung** statt, die sich an der bisherigen Anmeldeprozedur der Portalplattform orientiert. Als Besonderheit ist hier lediglich erwähnenswert, dass die Anmeldung hier mit dem Instanzieren eines charakteristischen Authentisierungsobjekts abgeschlossen wird. Dieses Hilfsobjekt der SWS-Architektur repräsentiert den vom Benutzer durchgeführten Authentisierungsmechanismus. Es wird später für eine authentisierungsbasierte Autorisierung verwendet (s. Kap. 4), bei der eine Autorisierung auch aufgrund der durchgeführten Authentisierung erfolgen kann. Bei jeder Anmeldung wird ein Objekt der Klasse **AuthenticationMechanism** an das *SSLSession*-Objekt geknüpft. Aufgrund der in diesem Objekt festgelegten unveränderlichen Eigenschaften kann zu einem späteren Zeitpunkt die Unterscheidung der durchgeführten Authentisierung erfolgen.

Die **Instanziierung des Umleitungstemplates** geschieht auf die in der Portalplattform übliche Weise.

Damit sind alle wichtigen Vorgänge während der Anmeldung erklärt. Die SWS-Architektur enthält eine weitere Klasse namens **Base64Decoder**. Ihre einzige Aufgabe ist die Umwandlung eines base64-kodierten Strings in einen menschlich lesbaren String, um die Credentials einer http-Authentisierung zu extrahieren. Ihre Funktionalität wird innerhalb des *HTTPSConnection*-Threads verwendet. Die Beschreibung der Kodierung ist aus www.freesoft.org/CIE/RFC/2065/56.htm entnommen.

Exkurs:

Die clientseitig verwendete Sicherheitseinrichtung spielt eine wichtige Rolle bei der Bewertung der Authentisierungsgüte. Wie bereits erwähnt, kann aufgrund der Gegebenheiten beim SSL-Handshake serverseitig allerdings lediglich das Zertifikat aber nicht seine Herkunft geprüft werden. Obwohl eine Unterscheidung durch die entsprechende Konfiguration des serverseitigen *Truststores* (vertrauenswürdige Stammzertifikate) vorgenommen werden kann, ist eine andere Möglichkeit, *X.509-Extensions* (Zertifikatsmerkmale) als Unterscheidungskriterium zu verwenden. Die besonderen Merkmale eines Zertifikats könnten dann ein Charakteristikum einer besonderen Sicherheitseinrichtung sein, und sie könnten innerhalb des betreffenden Handlers (im Verbindungsaufbau nachgelagerte Prüfung) oder in einem eigenen *TrustManager* (im Verbindungsaufbau vorgelagerte Prüfung, vgl. 2.3.4.3) verifiziert werden. Die Integrität dieser Zertifikatsmerkmale ist dabei genauso gesichert wie die übrige Information innerhalb des Zertifikats – nämlich durch die Signatur des Ausstellers.

3.3.3 Erkannte Lösungsdefizite

In diesem Abschnitt soll kurz auf identifizierte Probleme eingegangen werden, die im Zusammenhang mit der SWS-Architektur auftreten und im Rahmen dieser Arbeit nicht gelöst worden sind:

1. *Formulardaten, JavaScript und SSL*: Werden Formulardaten des Portals vom Client an den Server übertragen, so findet das clientseitig – wie bei der Portalplattform üblich – durch Aufruf einer JavaScript-Funktion statt. Obwohl für die Verbindung hierbei das SSL-Protokoll verwendet wird, und die Daten somit vertraulich übertragen werden, findet clientseitig im Browser eine Warnung statt, die den Benutzer vor einer nicht vertraulichen Übertragung seiner Daten warnt. Die Ursache hierfür ist allein durch den Aufruf einer JavaScript-Funktion begründet. Da die Verbindung das SSL-Protokoll verwendet, stellt dies kein Sicherheitsrisiko statt. Allerdings kann die fälschliche Browserwarnung den Benutzer irritieren.
2. *Konfiguration mehrerer SSL-Server*: Werden mehrere SSL-Server parallel gestartet, kommt es zu Konflikten, die nicht identifiziert werden konnten.

3.4 Zusammenfassung

Innerhalb dieses Kapitels ist die SWS-Architektur erarbeitet worden – der Teil der Sicherheitsarchitektur, der die Sicherheitskonzepte *Vertraulichkeit*, *Integrität* und *Authentisierung* innerhalb der Portalplattform umsetzt. Schrittweise ist mit Hilfe des SSL-Protokolls im Allgemeinen und der Java Secure Socket Extension (JSSE) im Speziellen der Lösungsansatz erläutert worden. Dabei ist ausführlich auf die Überlegungen, die bei dem Entwurf maßgebend gewesen sind, und auf die wesentlichen Aspekte des Entwurfs eingegangen worden. Darüber hinaus sind die wichtigsten Schwierigkeiten und deren Überwindung bei der Implementierung sowie erkannte Probleme erklärt worden.

4 Autorisierung

In diesem Kapitel wird die Umsetzung des Sicherheitskonzeptes *Autorisierung* bei der Portalplattform erläutert. Dabei wird gemäß den Ausführungen in Abschnitt 2.1.2.2 zunächst auf konzeptioneller Ebene definiert, welche schützenswerte Ressourcen mit Zugriffsrechten zu versehen sind und wie mit diesen Rechten umgegangen werden kann (Definition, Zuordnung, Kombination, etc.). Im Anschluss wird erläutert wie die Autorisierungsarchitektur aufgebaut ist. Abschließend wird auf Detailfragen zur Initialisierung von Zugriffsrechten eingegangen.

4.1 Konzeption

In Abschnitt 2.1.2.2 ist bei der Erläuterung des Sicherheitskonzeptes *Autorisierung* in abstrakter Weise von **Rechten** und zu schützenden **Ressourcen** gesprochen worden. Für eine Umsetzung ist es erforderlich, diese Sichtweise zu konkretisieren und auf die Gegebenheiten bei der Portalplattform anzupassen.

Betrachtet man das Schichtenmodell der Portalplattform (vgl. Abb. 1) und vergegenwärtigt man sich die Ausführungen in Abs. 2.2, so kann man unmittelbar zwei schützenswerte Arten von Ressourcen erkennen: die Anwendungslogik, die durch die **Handler** verkörpert wird, und die persistente Information, die durch die **Assets** verkörpert wird. Schützt man die Handler vor nicht autorisiertem Zugriff, so werden die festgelegten Abläufe geschützt. Nur wenn ein Vorgang im jeweiligen Kontext autorisiert ist, kann somit eine bestimmte Handlung durchgeführt werden. Schützt man die Assets vor nicht autorisiertem Zugriff, so werden gespeicherte Informationsinhalte geschützt. Nur wenn der Informationszugriff im jeweiligen Kontext autorisiert ist, kann die gewünschte Information eingesehen oder ggf. verändert werden.

Es sei an dieser Stelle ergänzend angemerkt, dass bereits eine Autorisierung auf Präsentationsebene existiert, die innerhalb der Templates mit Hilfe von bedingten Platzhaltern stattfindet. Die zu schützenden Ressourcen sind hierbei die anzuzeigenden oder nicht anzuzeigenden Verknüpfungen. Dies wird später weiter ausgeführt, da dies nicht Gegenstand der Konzeption ist.

Als Gegenstand dieser Arbeit wird der Zugriffsschutz der Ressource **Handler** vereinbart. Die Autorisierung auf Asset-Ebene ist somit ausdrücklich nicht Gegenstand dieser Arbeit⁴⁸. Diese Aussage legt zunächst fest, für welche Ressourcen Zugriffsrechte definiert werden sollen.

Nachdem festgelegt ist, welche Ressourcen zu schützen sind, werden als nächstes die Rechte betrachtet. Die festzulegenden Rechte sollen bestimmten Kriterien genügen. Dies kann als Anforderungsprofil für die Gestaltung der Rechte verstanden werden.

⁴⁸ Notiz P. Hupe: Da die Prozesse (Handler) den Objekten (Assets) übergeordnet sind und ein Zugriff auf die Objekte, was das Web Interface angeht, nur über die Prozesse möglich ist, impliziert ein Schutz der Prozesse einen Schutz der Objekte. Hierbei existieren zwei Einschränkungen:

- Fehler in den Handlern erlauben einen Zugriff auf Objekte, der nicht gestattet wäre.
- Schnittstellen, die nicht die Handler verwenden (WebDAV, WebServices), nutzen nicht diese Autorisierung.

4.1.1 Anforderungen

Zunächst einmal soll ein Recht durch ein sog. **Prädikat** in deklarativer Weise festgelegt werden. Unter dem Begriff *Prädikat* ist hier eine allgemeinverständliche Bezeichnung bzw. Charakterisierung eines Rechts zu verstehen, die zusätzlich in der Lage sein soll, Parameter aufzunehmen, die das spezielle Recht spezifizieren.

Ein einfaches Beispiel für ein solches *Prädikat* wäre die Bezeichnung **isMemberOf (Administratoren)**. In diesem Beispiel ist der Schlüsselbegriff *isMemberOf* als Charakteristikum des *Prädikats* und die Bezeichnung *Administratoren* als Spezifizierung (Parameter) des Rechts zu erkennen. Diese Bezeichnung ist insofern allgemeinverständlich, als dass eine Ressource (ein Handler), die durch dieses *Prädikat* geschützt wäre, lediglich für Mitglieder einer Administratorengruppe zugänglich wäre.

Diese *Prädikate* werden im jeweiligen Kontext, in dem sie ausgeführt werden, zu einem booleschen Wert evaluiert. Dies ist so zu interpretieren, dass für den Fall, dass das *Prädikat* im ausgeführten Kontext zum booleschen Wert *true* evaluiert, die Autorisierung vom System erteilt wird und im Fall, dass das *Prädikat* zum booleschen Wert *false* evaluiert, die Autorisierung vom System versagt wird.

Für das genannte Beispiel würde das *Prädikat isMemberOf (Administratoren)* genau dann zu *true* evaluieren, wenn der Benutzer der Session sich innerhalb der Gruppe der Administratoren befindet. Für Benutzer, die sich nicht in dieser Gruppe befinden, würde das *Prädikat* zu *false* evaluieren.

Es soll die Möglichkeit bestehen, vorhandene *Prädikate* in deklarativer Weise logisch miteinander zu kombinieren und hieraus neue, komplexere *Prädikate* zu generieren. Hierbei sollen die *Prädikate* zunächst durch die booleschen Operatoren NOT, AND und OR miteinander kombiniert sein. Wird ein zusammengesetztes *Prädikat* innerhalb eines Kontextes evaluiert, dann ist das Ergebnis, die boolesche Kombination der in diesem Kontext evaluierten *Prädikate*, aus denen das zusammengesetzte *Prädikat* besteht. Die rekursive Definition von *Prädikaten* muss dabei ausgeschlossen sein.

In Abschnitt 2.1.2.2 ist angedeutet worden, dass die Voraussetzungen für die Erfüllung von Rechten auf unterschiedlichen Arten von Eigenschaften beruhen können. Die Eigenschaften eines *Prädikats* sind durch die Parameter, die das *Prädikat* aufnimmt, und durch seine programmatische Festlegung bestimmt. Elementare *Prädikate*⁴⁹ können dabei aufgrund der Art von Eigenschaften oder Merkmalen, die sie berücksichtigen, unterschieden werden. In dieser Arbeit soll die Unterscheidung aufgrund persönlicher, ressourcenbasierter und authentisierungsbasierter Merkmale vorgenommen werden. Darüber hinaus sind weitere Kategorien denkbar, die hier nicht weiter betrachtet werden. *Prädikate*, deren Erfüllungsvoraussetzungen auf **persönlichen** Merkmalen beruhen, betrachten bei ihrer Evaluierung die Identität des betreffenden Benutzers. *Prädikate*, deren Erfüllungsvoraussetzungen auf **ressourcenbasierten** Merkmalen beruhen, betrachten bei ihrer Evaluierung die Zustände oder Eigenschaften der verwendeten Ressourcen (Assets). *Prädikate*, deren Erfüllungsvoraussetzungen auf **authentisierungsbasierten** Merkmalen beruhen, betrachten bei ihrer Evaluierung den Mechanismus, der zur Authentisierung der Identität eines Benutzers geführt hat.

⁴⁹ also solche, die nicht zusammengesetzt sind

Die Zuordnung der *Prädikate* zu ihren zu schützenden Ressourcen (Handler) soll auf deklarative Weise erfolgen. Darüber hinaus sollen Autorisierungen, die auf einer grundlegenden Projektstufe erteilt worden sind, nicht mehr auf einer darüberliegenden Projektstufe veränderbar sein. Diese letzte Forderung regelt die Verantwortlichkeit für die Erteilung von Autorisierungen.

Bei Ressourcen (Handler), denen kein *Prädikat* zugeordnet ist, sollen mindestens zwei alternative Strategien existieren, die die Autorisierung regeln. Im ersten Fall soll der Zugriff auf die Ressource (Handler) stets gewährt werden. Diese Strategie wird als *liberal* bezeichnet. Sie ist für die Migration von einem System ohne Autorisierungskonzept zu einem System mit Autorisierungskonzept sinnvoll. Im zweiten Fall soll der Zugriff hingegen stets versagt werden. Diese Strategie wird als *restriktiv* bezeichnet. Sie ist für ein System mit Autorisierungskonzept, das kontinuierlich um neue Anwendungslogik erweitert wird, sinnvoll.

Es soll nachträglich nachvollzogen werden können, welche *Prädikate* in einem Ausführungskontext existent und welchen Ressourcen (Handlern) sie jeweils zugeordnet gewesen sind. Darüber hinaus ist von Interesse in Falle des Zugriffs auf eine Ressource (Handler), welcher Benutzer den Zugriff getätigt hat und ob dieser gewährt oder versagt worden ist. Diese Funktionalität wird unter dem Begriff *Auditing* zusammengefasst.

4.2 Entwurf der Autorisierungsarchitektur

Nachdem das Anforderungsprofil für die Autorisierungsarchitektur festgelegt worden ist, sind die Maßnahmen zur Definition, Zuordnung und Auswertung von Zugriffsrechten festzulegen. In diesem Kontext werden die Zugriffsrechte als **Berechtigungen (Authorities)** bezeichnet. Die zuvor betonten *Prädikate* sind dabei deklarative Repräsentationen von programmatisch implementierten Berechtigungen.

4.2.1 Die Berechtigung

Das zentrale Objekt der Autorisierungsarchitektur ist die Berechtigung (*Authority* – vgl. Abb. 23). Unter einer Berechtigung ist ein Objekt zu verstehen, das im jeweiligen Kontext, in dem es ausgeführt wird, zu einem boolschen Wert evaluiert werden kann. Zu diesem Zweck besitzt das Berechtigungsobjekt eine Methode *check*, dessen Rückgabewert vom Typ *boolean* sein soll.

Es sind grundsätzlich zwei Arten von Berechtigungen möglich: elementare Berechtigungen und zusammengesetzte Berechtigungen. Die elementaren Berechtigungstypen sind dabei gemäß den Merkmalen, die sie abprüfen, zu kategorisieren. Gemäß den zuvor dargestellten Kategorien existieren **subjektbasierte**, **objektbasierte** und **authentisierungsbasierte** Berechtigungen.

Eine elementare Berechtigung ist eine Instanz eines programmatisch festgelegten Berechtigungstyps. Jeder Berechtigungstyp implementiert dabei die Methode *check* auf eine für ihn charakteristische Weise. Darüber hinaus ist jedem elementaren Berechtigungstyp ein Begriff zugeordnet, durch das *Prädikate* den jeweiligen Berechtigungstyp, der durch sie repräsentiert wird, spezifizieren sollen.

Eine zusammengesetzte Berechtigung repräsentiert einen boolschen Ausdruck von elementaren oder zusammengesetzten Berechtigungen. Da die Architektur die Implementierung der drei elementaren boolschen Operationen NOT, AND und OR vorsieht, existieren demnach drei zusammengesetzte Berechtigungstypen, die je eine dieser Operationen repräsentieren. Die Aufgabe der Methode *check* bei einer zusammengesetzten Berechtigung beschränkt sich hierbei darauf, die Methode(n) *check* rekursiv aufzurufen und die Ergebnisse in entsprechender, logischer Weise zu einem Resultat zu kombinieren.

Abb. 23 zeigt die abstrakte Klasse **Authority**, die eine Generalisierung einer jeden Berechtigung darstellt. Dem untergeordnet sind die drei abstrakten Klassen, die zur Kategorisierung der elementaren Berechtigungstypen dienen: **SubjectbasedAuthority**, **ObjektbasedAuthority** und **AuthenticationbasedAuthority**. Elementare Berechtigungstypen sind in der genannten Abbildung nicht sichtbar. Ebenfalls der Klasse *Authority* untergeordnet sind die drei Klassen für zusammengesetzte Berechtigungen zu sehen, die jeweils eine der drei zu implementierenden Operationen repräsentieren: **NotAuthority**, **AndAuthority** und **OrAuthority**.

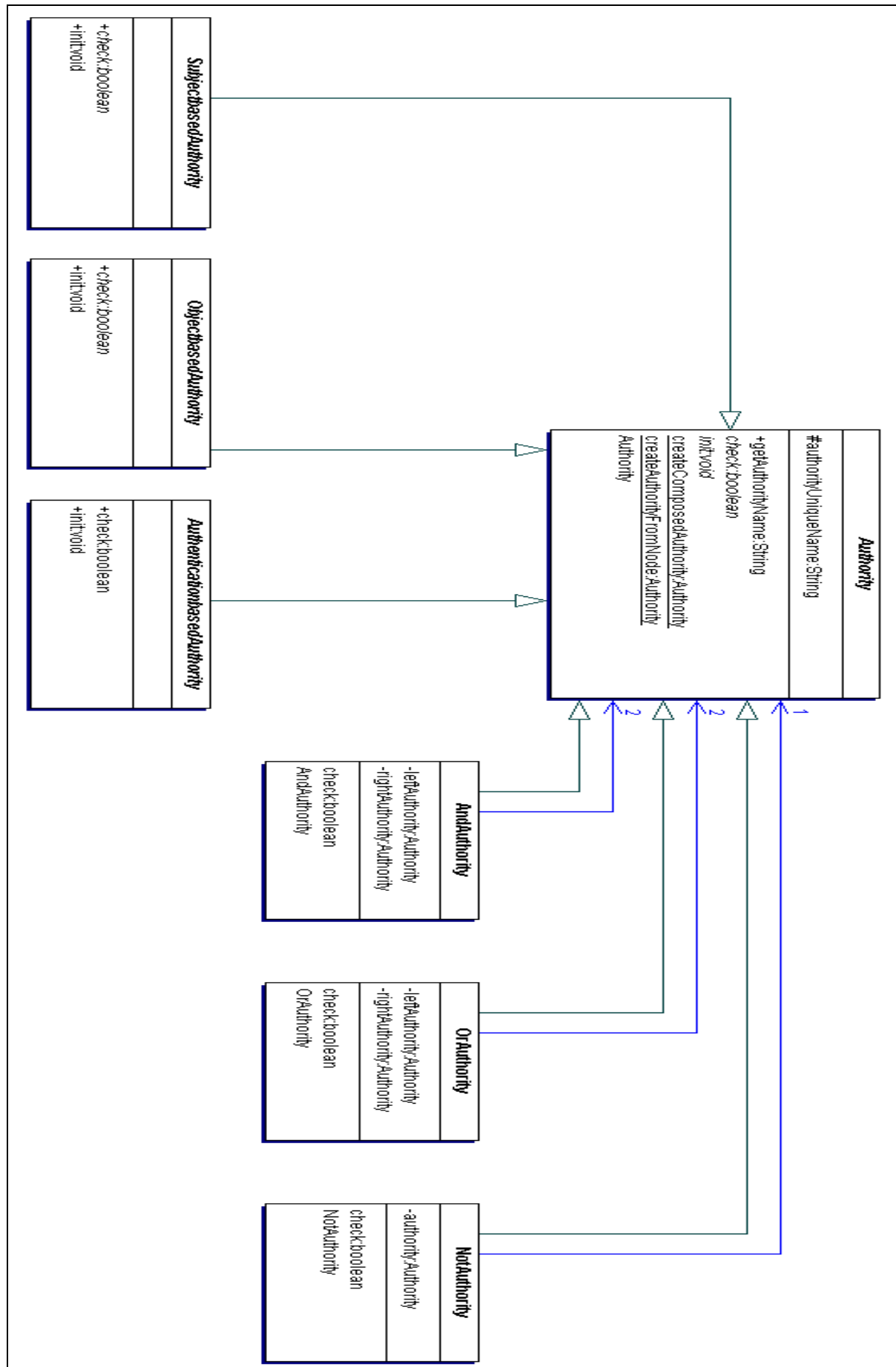


Abb. 23: Autorisierungsarchitektur (Berechtigungen)

4.2.2 Die Definition von Berechtigungen

Innerhalb der Autorisierungsarchitektur erfolgt die Definition von Berechtigungen teilweise programmatisch und teilweise deklarativ. Dies soll nun näher erläutert werden.

4.2.2.1 Programmatische Definition der Berechtigungen

In Abschnitt 4.2.1 ist von elementaren Berechtigungen gesprochen worden, die durch Instanzen eines Berechtigungstyps verkörpert werden. Jeder Berechtigungstyp implementiert die Methode *check* auf eine für ihn charakteristische Weise, wobei typischerweise ein Vergleich zwischen den Berechtigungsattributen, die die Berechtigung bei einem Initialisierungsvorgang während der Instanziierung erhält, und Eigenschaften, die dem Ausführungskontext entnommen werden, stattfindet. Jeder Berechtigungstyp hat hierbei eine für ihn typische *Prüfaufgabe* durchzuführen. Typische Prüfaufgaben sind die Prüfung, ob es sich bei dem Benutzer um eine bestimmte Person handelt, ob der Benutzer Mitglied in einer festgelegten Gruppe ist, etc.

Abb. 24 zeigt die in dieser Arbeit festgelegten, subjektbasierten Berechtigungstypen: Sie dienen als Beispiel, um die programmatische Festlegung von Berechtigungen zu erläutern. Neben den genannten Berechtigungstypen findet man in der Abbildung das Personen-, Gruppen- sowie Rollenmodell der Plattform-Architektur.

- Die personenbasierte Berechtigung (***PersonbasedAuthority***) prüft, ob es sich bei dem betreffenden Benutzer (Ausführungskontext) um eine ganz bestimmte Person handelt (Attribut der Berechtigung). Hierdurch können Ressourcen einzelnen Individuen zugänglich gemacht werden, während sie für alle anderen Benutzer unzugänglich sind.
- Die gruppenbasierte Berechtigung (***GroupbasedAuthority***) prüft, ob der betreffende Benutzer (Kontext) eine Mitgliedschaft in einer festgelegten Gruppe (Attribut) hat. Hierdurch können Ressourcen Gruppen zugeordnet werden, während sie für andere Gruppen unzugänglich sind.
- Die rollenbasierte Berechtigung (***RolebasedAuthority***) prüft, ob der betreffende Benutzer (Kontext) eine Mitgliedschaft in einer festgelegten Gruppe und hierin eine festgelegte Rolle besitzt (Attribute). Hierdurch können Zugriffsrechte auf der Basis von Rollen in Gruppen definiert werden.
- Die kategoriebasierte Berechtigung (***CategorybasedAuthority***) prüft, ob der betreffende Benutzer (Kontext) eine Mitgliedschaft in mindestens einer Gruppe der festgelegten Kategorie (Attribut) besitzt. Hierdurch können Zugriffsrechte für Gruppenkategorien festgelegt werden.

Wie unschwer zu erkennen ist, können weitere Berechtigungstypen implementiert werden, die weitere spezifische Autorisierungskonzepte umsetzen und die beliebig komplex sein können. Es ist hierbei auf eine aussagekräftige Kategorisierung der Berechtigungstypen gemäß Abb. 23 zu achten. Für objektbasierte und authentisierungsbasierte Berechtigungen sind im Rahmen dieser Arbeit ebenso spezielle Berechtigungstypen entstanden. Beispiele für authentisierungsbasierte Berechtigungstypen sind die ***PasswordbasedAuthenticationAuthority***, ***CertificatebasedAuthenticationAuthority*** und ***SmartCardbasedAuthenticationAuthority***, die eine Autorisierung auf der Basis unterschiedlicher Authentisierungsmechanismen ermöglichen. Als Beispiel für einen objektbasierten Berechtigungstyp dient die ***AdministratorDocumentAuthority***.

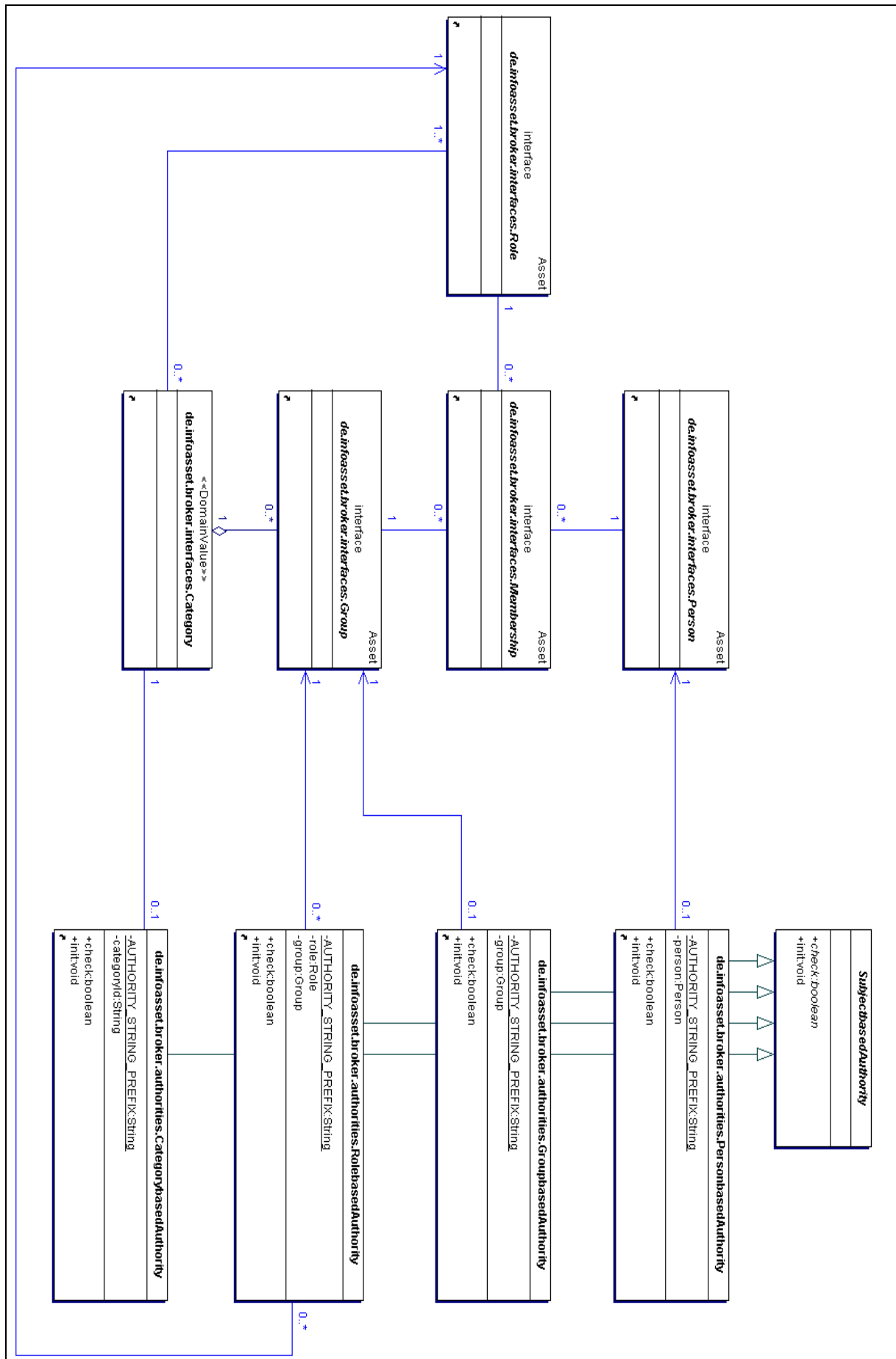


Abb. 24: Autorisierungsarchitektur (Subjektbasierte Berechtigungen)

4.2.2.2 Deklarative Definition der Berechtigungen

Während die programmatische Definition von Berechtigungstypen festlegt, welche Arten von elementaren Berechtigungen möglich sind, legt die deklarative Definition die tatsächlich vorhandenen Berechtigungen fest. Hierbei spielt analog zur *handler.txt* eine andere Textdatei eine wichtige Rolle: die ***authorities.txt***. Diese Textdatei befindet sich im selben Verzeichnis wie die entsprechende *handler.txt*. In ihr findet eine Zuordnung von Prädikaten zu Berechtigungstypen statt.

Die *authorities.txt* ist die Textdatei, in der deklarativ die verwendbaren Prädikate definiert werden. Hierbei kann auf jeder Projektebene der Portalplattform die Definition neuer Prädikate stattfinden. Eine Redefinition von bereits auf untergeordneten Projektebenen definierten Prädikaten ist nicht gestattet. Auf diese Weise können bestehende Rechte nicht auf höheren Projektebenen liberalisiert und die Autorisierungsarchitektur so ausgehebelt werden. Ähnlich der *handler.txt* existiert somit im Stammverzeichnis eines jeden Projekts eine *authorities.txt*, in der die auf der betreffenden Projektebene neuen Prädikate festgelegt werden. Prädikate, die in grundlegenden Projekten definiert worden sind, können auch auf höheren Projektebenen wieder verwendet werden.

Innerhalb einer *authorities.txt* existieren zwei Arten von Definitionen. Es werden zunächst diejenigen Prädikate deklariert, die auf elementaren Berechtigungen aufbauen. Hierbei wird folgende allgemeingültige Syntax verwendet:

Schlüsselwort(Parameter1, Parameter2, ..., ParameterN) = voller Klassenname der Berechtigung

Das Schlüsselwort und die Art und Anzahl der einzutragenden Parameter werden durch die Berechtigungsklasse festgelegt (Berechtigungstyp). Es liegt in den Händen eines Entwicklers das Schlüsselwort sowie die Art und Anzahl der Parameter festzulegen. Ein typisches Beispiel für die Deklaration eines elementaren Prädikats wäre

isPerson("perera@tu-harburg.de") = de.infoasset.broker.authorities.PersonbasedAuthority

Deklarierte, elementare Prädikate können fortan verwendet werden. Dies gilt insbesondere für die Verwendung in der zweiten Art von Definitionen – den zusammengesetzten Prädikaten. In diesen Definitionen wird ein beliebiger Prädikatsname festgelegt, dem ein boolescher Ausdruck zugeordnet wird. Dieser Ausdruck besteht aus den booleschen Operationen NOT, AND und OR, die in genau dieser Schreibweise zu verwenden sind. Innerhalb eines Ausdrucks hat die Operation NOT die höchste und die Operation OR die geringste Bindungsstärke. Die Ausdrücke können durch Klammerung mit den Klammern "[" und "]" beliebig verschachtelt sein. Die Operanden eines logischen Ausdrucks sind hierbei selbst Prädikate – elementare oder zusammengesetzte. Wichtig bei der Verwendung eines Prädikats innerhalb eines Ausdrucks ist, dass das Prädikat auf der betreffenden Projektebene bereits deklariert ist, d.h. es dürfen keine Prädikate aus übergeordneten Projektebenen verwendet werden, da die übergeordneten Projekte aus Sicht eines grundlegendes Projekts austauschbar sein müssen. Ein einfaches Beispiel für ein zusammengesetztes Prädikat wäre

alleAusserCesar = NOT [isPerson("perera@tu-harburg.de")]

Natürlich können die booleschen Ausdrücke wesentlich komplexer als das hier genannte Beispiel sein. Der Ausdruck *alleAusserCesar* könnte nun in einer anderen zusammengesetzten Prädikatsdefinition weiterverwendet werden.

Im Rahmen dieser Arbeit ist eine *authorities.txt* als Referenz umgesetzt worden. Sie ist im Projektverzeichnis „broker“ zu finden. Dies ist die grundlegende Projektebene für sämtliche Projekte, die mit Hilfe der Portalplattform erstellt werden. Die Referenzdatei enthält als Kommentar eine ausführliche Beschreibung über die Syntax und über die Art und Weise, wie diese zu verwenden ist.

4.2.3 Die Zuordnung von Berechtigungen zu Handlern

Nachdem die Definition von Berechtigungen ausführlich erläutert worden ist, wird die Zuordnung von Zugriffsrechten (Berechtigungen) zu Ressourcen (Handlern) betrachtet.

4.2.3.1 Fachlicher Entwurf

Für jedes deklarierte Prädikat – elementar oder zusammengesetzt – wird eine Berechtigung instanziiert, die innerhalb eines Ausführungskontextes zu einem booleschen Wert evaluiert werden kann. Hieraus resultiert, dass Prädikate die deklarative Repräsentation von vorhandenen Berechtigungen bilden. Um eine gewünschte Ressource (Handler) mit einem Zugriffsrecht (Berechtigung) zu versehen, muss das Zugriffsrecht durch ein Prädikat festgelegt worden sein.

Die Zuordnung ist aus Anwendungssicht sehr simpel. Innerhalb der betreffenden *handler.txt*, die für das eigene Projekt maßgeblich ist, findet die Zuweisung in deklarativer Weise statt. Hierzu wird dem technischen Klassennamen ein Doppelpunkt ":" angefügt, gefolgt von dem Prädikat, das die entsprechende Berechtigung repräsentiert. Die allgemeine Syntax hat somit folgende Gestalt:

<Kategorie/Aktion>=<voller Klassenname>:<Prädikat>

Diese Syntax wird noch durch ein Beispiel verdeutlicht:

```
directories/default=de.infoasset.broker.handler.directories.DefaultHandler:isPasswordAuthentication()
```

Hieraus ist ersichtlich, dass der *DefaultHandler* für Verzeichnisse nur zugänglich ist, wenn der Benutzer eine Passwort-Authentisierung durchgeführt hat. Dieses Beispiel ist zwar nicht sonderlich sinnvoll, zeigt allerdings, wie einfach die deklarative Zuordnung stattfindet.

Bei dieser Zuweisung sind zwei Dinge sicherzustellen: Zunächst muss das zugewiesene Prädikat auf der betreffenden Projektebene oder darunter deklariert worden sein (s. 4.2.2). Außerdem darf dem betreffenden Handler nicht schon auf einer grundlegenden Projektebene ein Prädikat zugewiesen worden sein. Diese letzte Forderung stellt sicher, dass die Autorisierungsarchitektur nicht durch eine Neuordnung der Rechte auf einer Ebene, die hierfür keine Kompetenz hat, ausgehebelt wird.

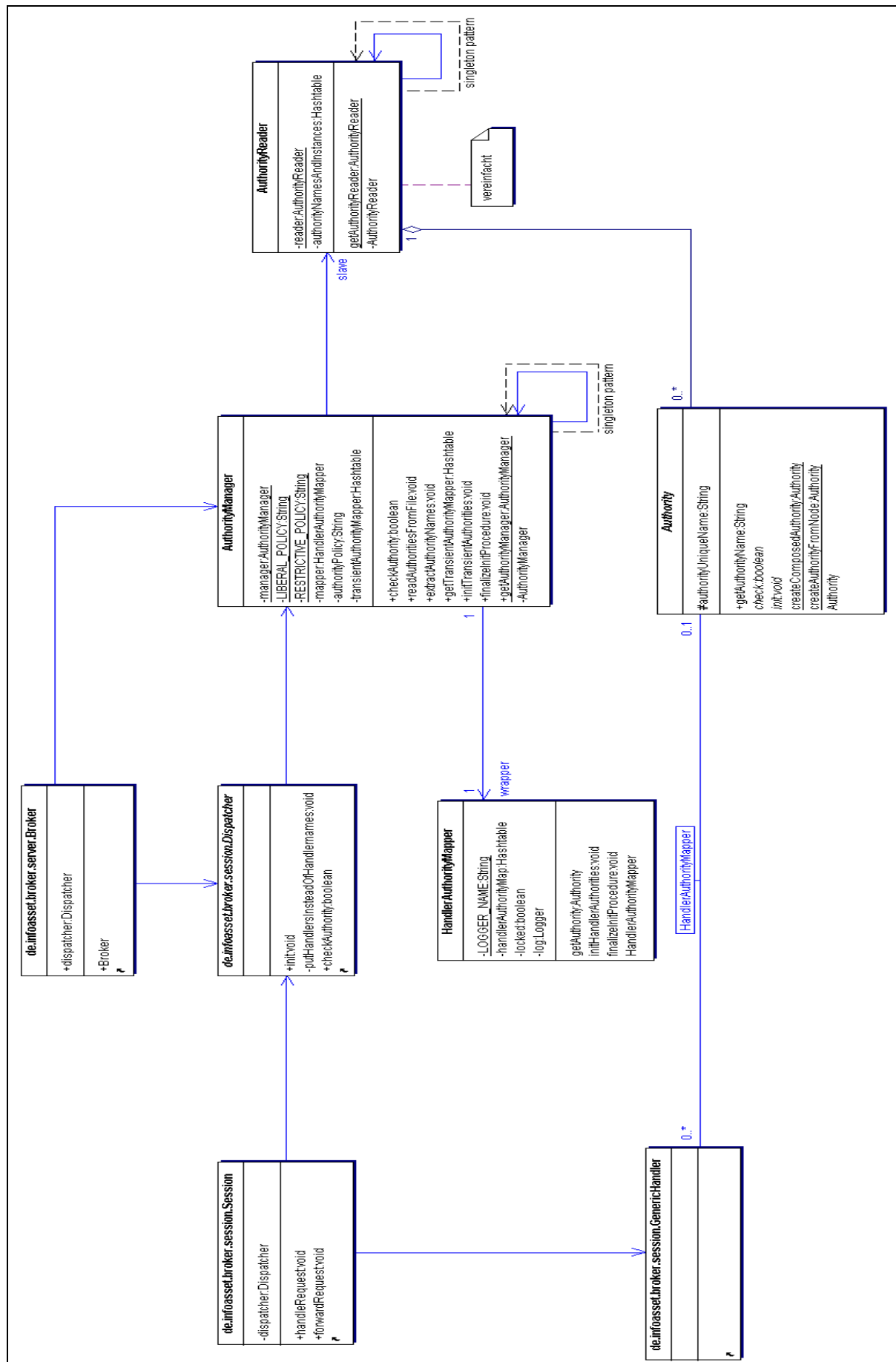


Abb. 25: Autorisierungsarchitektur (Organisationseinheiten)

4.2.3.2 Technischer Entwurf

Während die deklarative Zuordnung der Zugriffsrechte sehr einfach und anwendungsfreundlich ist, ist der programmatische Teil der Architektur, der sich dahinter verbirgt umso komplexer. Abb. 25 zeigt, wie die Zuordnung von Handlern, die hier durch die generalisierte Klasse *GenericHandler* repräsentiert werden, zu Berechtigungen (*Authorities*) programmatisch realisiert wird. Die Aufgabe der Zuordnung übernimmt hierbei die Klasse ***HandlerAuthorityMapper***, die die Zuordnungstabelle enthält. Nach Abschluss des Initialisierungsvorgangs kann die zu einem Handler gehörige Berechtigung durch Aufruf der Methode ***getAuthority (GenericHandler handler)*** stets programmatisch erfragt werden. Der eigentliche Initialisierungsvorgang der Zuordnungstabelle, der die deklarative Zuordnung in der *handler.txt*⁵⁰ in eine programmatisch nutzbare Zuordnung in der Zuordnungstabelle vornimmt, ist hierbei der komplizierteste Ablauf. Er wird in Abschnitt 4.3 erläutert.

4.2.4 Die Auswertung von Berechtigungen

Das Sicherheitskonzept *Autorisierung* sieht vor, dass vor dem Zugriff auf eine Ressource das entsprechende Zugriffsrecht im jeweiligen Kontext auf Erfüllung hin geprüft wird. Zunächst einmal ist es also wichtig zu wissen, von welcher Stelle aus auf die Funktionalität eines Handlers zugegriffen wird. Innerhalb der Architektur der Portalplattform findet dieser Zugriff auf einen Handler vom *Session*-Objekt, das den jeweiligen Ausführungskontext repräsentiert, aus statt. Das *Session*-Objekt ruft bei dem betreffenden Handler die Methode *handleRequest* auf, die den Handler somit ausführt. Es ermittelt dabei den betreffenden Handler, indem es vorher beim *Dispatcher*-Objekt anfragt. Der genannte Aufruf findet innerhalb des *Session*-Objekts an zwei Stellen statt: innerhalb der Methoden ***handleRequest*** und ***forwardRequest***. Der Programmcode für die Ermittlung des Handlers und seine Ausführung ist in beiden Fällen identisch:

```
GenericHandler h = dispatcher.getHandler (documentId);  
...  
h.handleRequest (this, output, documentId);
```

Das Schlüsselwort ***this*** bezieht sich hierbei auf das *Session*-Objekt, welches den Handler aufruft. Die genannten Stellen im Programmcode sind somit die Stellen, an denen eine Berechtigungsprüfung stattzufinden hat und die in entsprechender Weise abzuändern sind. Die Bedeutung des *Session*- und des *Dispatcher*-Objekts für die Autorisierungsarchitektur sind der Grund dafür, dass diese Klassen in Abb. 25 wieder zu finden sind.

Nachdem die wesentlichen Stellen für die Autorisierungsprüfung ermittelt sind, wäre es ein Einfaches, die zu dem jeweiligen Handler gehörige Berechtigung zu ermitteln (s. 4.2.3) und die erforderliche Prüfung durchzuführen. Die Autorisierungsarchitektur weist diesbezüglich allerdings zwei Besonderheiten auf:

⁵⁰ bzw. den *handler.txt*-Dateien

Zunächst einmal führt der Weg zur Autorisierungsarchitektur stets über den **AuthorityManager**, der sämtliche administrative Aufgaben dieser Komponente übernimmt bzw. weiterdelegiert. Dieses Objekt ist gemäß dem *Singleton-Pattern* zu implementieren. In diesem konkreten Fall stellt der *AuthorityManager* die Methode **checkAuthority** zur Verfügung, in der mit Hilfe der Zuordnungstabelle (s. 4.2.3) die betreffende Berechtigung ermittelt und diese auf Erfüllung hin – anhand ihrer Methode **check** – geprüft wird. **Somit findet die Auswertung der Berechtigung innerhalb der Methode checkAuthority statt, die die eigentliche Prüfung durch Delegation an die entsprechende Methode check durchführt.** Darüber hinaus werden innerhalb der Methode *checkAuthority* die in Abs. 4.1 beschriebenen alternativen Strategien (liberal, restriktiv, etc.) implementiert.

Eine weitere untergeordnete Besonderheit liegt im Aufruf der Methode *checkAuthority*. Diese könnte an den ermittelten Prüfstellen prinzipiell direkt vom *Session*-Objekt aus aufgerufen werden, da der *AuthorityManager* ja ein Singleton ist. Diesem Aufruf zwischengeschaltet ist allerdings das *Dispatcher*-Objekt, das eine gleichnamige Methode implementiert und den Aufruf lediglich an den *AuthorityManager* weiterdelegiert. Die Berechtigung für diesen scheinbaren Umweg liegt in der Tatsache begründet, dass hierdurch die Zugriffspfade auf die Autorisierungsarchitektur klarer sind. Der *Dispatcher* muss während des Initialisierungsvorgangs der Zuordnungstabelle mehrmals auf den *AuthorityManager* zugreifen, so dass durch diese Maßnahme die Anzahl der direkten Aufrufer geringer gehalten wird. Darüber hinaus findet in der zwischengeschalteten Methode des *Dispatchers* das *Logging* statt, das festhält, wer den Zugriffsversuch unternimmt und ob der Zugriff gewährt oder versagt wird.

Nachdem die Auswertungslogik von Berechtigungen ausführlich erläutert ist, werden die genannten Veränderungen innerhalb des *Session*-Codes kurz skizziert:

```
GenericHandler h = dispatcher.getHandler(documentId);
...
if (this.dispatcher.checkAuthority (h, this, documentId))
    h.handleRequest (this, output, documentId);
else
    this.forwardRequest (output, "security/accessDenied.htm");
```

Man erkennt, dass für den Fall, dass die Berechtigungsprüfung erfolgreich ist, das System wie bisher reagiert. Ein abweichendes Verhalten erhält man für den Fall, dass die Berechtigungsprüfung misslingt. Im letzteren Fall wird ein Handler ausgeführt, der ein Template instanziiert, das den Benutzer über die Zugriffsverletzung informiert (JavaScript → Alert) und die zuvor angezeigte Seite erneut anzeigt (History → Back).

4.2.5 Die Gesamtarchitektur

Nachdem auf sämtlich wichtige Aspekte der Autorisierungsarchitektur eingegangen worden ist, wird diese, um das Bild zu vervollständigen, als Ganzes dargestellt (s. Abb. 26). Die Hilfsklassen und die Ausnahmeobjekte (*Exceptions*), die für die Implementierung erforderlich sind, sind in dieser Darstellung fortgelassen.

Die Bedeutung der Klassen *Broker* und *AuthorityReader* wird im Zusammenhang mit der Initialisierung der Zuordnungstabelle noch erläutert.

4.3 Details der Implementierung

Der vorangegangene Abschnitt über den Entwurf (4.2) hat die Berechtigung als Verkörperung des Zugriffsrechts auf Handler des infoAsset Broker herausgehoben. Es ist dargestellt worden, wie Berechtigungen sowohl deklarativ als auch programmatisch festgelegt werden. Hierbei ist zunächst offen geblieben, wie die deklarative Definition von Prädikaten in eine operative Instanziierung von Berechtigungen umgewandelt wird. Im vorangegangenen Abschnitt ist auch behandelt worden, dass die operative Zuordnung von Berechtigungen zu Handlern mit Hilfe einer Zuordnungstabelle stattfindet. In diesem Zusammenhang ist offen geblieben, wie die deklarative Zuordnung in der *handler.txt* in eine operative Zuordnungstabelle abgebildet wird. Diese beiden Fragen sind eng miteinander verknüpft und werden nun durch die Beschreibung des Initialisierungsvorgangs erläutert.

4.3.1 Die Initialisierung der Berechtigungen

Eine besondere Eigenschaft, die im Rahmen der Initialisierung eine wichtige Rolle spielt, ist die Tatsache, dass die deklarative Zuordnung von Berechtigungen zu Handlern in der *handler.txt* erfolgt. Darüber hinaus werden in der Architektur der Portalplattform die Dateien aller Projektebenen berücksichtigt. Dies äußert sich programmatisch in der Tatsache, dass über die Dateien der verschiedenen Projektebenen iteriert wird, um Zuordnungen von *documentIds* zu Handlern bereitzustellen. Für diese Zuordnung hat der *Dispatcher* die Verantwortung. Da die betreffenden Dateien innerhalb der Sicherheitsarchitektur ebenfalls für die Zuordnung von Prädikaten verwendet werden und da die bisherige Funktionalität der Portalplattform für die Zuordnung von *documentIds* zu Handlern weiterhin verwendet werden soll, ist es erforderlich, dass die jeweiligen Zuordnungen von Berechtigungen zu Handlern aus den String-Repräsentationen entfernt und für eine spätere Verwendung zwischengespeichert werden. Darüber hinaus wird in jedem Iterationsschritt die *handler.txt* einer Projektebene abgearbeitet, so dass dieser Mechanismus ebenfalls zum Abarbeiten der betreffenden *authorities.txt* verwendet werden kann.

Um diese komplexen Sachverhalte näher zu erläutern, werden hier wichtige Codefragmente aufgeführt. Diese Initialisierungsprozedur ist Bestandteil des Konstruktors der Klasse *Broker*.

```
// Initialisierung der Zuordnung von documentIds zu Handlern
//
roots = new String [tok.countTokens()];
for (int i=0; i<roots.length; i++) {
    roots[i] = home + File.separator + "projects" + File.separator +
        tok.nextToken();
    log.debug ("broker: root[" + i + "]= " + roots[i]);
}
// roots [i] enthält die Namen der jeweiligen Projektpfade
// i∈{0,..., Anzahl der Projektebenen - 1}
...
if (automaticHandlerReload)
    this.dispatcher = new ReloadingDispatcher (services);
else
    this.dispatcher = new SimpleDispatcher (services);
dispatcher.init ();
// definiert den zu verwendenden Dispatcher (Dispatcher ist abstrakt)
// und initialisiert diesen (vererbte Methode der abstrakten Klasse)
```

```
for (int i=roots.length-1; i>=0; i--) { // start at broker
    dispatcher.init (roots[i] + File.separator + "handler.txt");
}
// iteriert über die Projektebenen, wobei der Dispatcher mit der
// handler.txt der jeweiligen Projektebene initialisiert wird
```

Die Iteration über die verschiedenen Projektebenen findet in der letzten for-Schleife statt. Bis dahin sind somit keine Änderungen erforderlich, da die bisherige Zuordnung von *documentIds* zu Handlern unverändert fortbestehen soll. Um den Iterationsmechanismus für die eigene Architektur zu nutzen (Instanziierung der Berechtigungen aus der *authorities.txt*), ist der Konstruktor der Klasse *Broker* innerhalb der for-Schleife und ggf. darüber hinaus zu editieren. Um auf die veränderten Gegebenheiten im Zusammenhang mit der *handler.txt* einzugehen (Entfernen und Zwischenspeichern der Berechtigung-Handler-Zuordnungen), ist die während der Iteration aufgerufene Methode *init (String handlerTable)* im *Dispatcher* zu editieren. Diese Kausalitäten rechtfertigen, dass die Klassen *Broker* und *Dispatcher* in die Autorisierungsarchitektur eingebunden worden sind (vgl. Abb. 25).

Nachdem skizziert ist, an welchen Stellen im Programmcode die Autorisierungsarchitektur prozedural anzusetzen ist, sollen die erforderlichen Veränderungen detaillierter analysiert werden. Der erste Schritt ist hierbei auf jeder Projektebene die Instanziierung der Berechtigungen, die deklarativ durch Prädikate in der *authorities.txt* festgelegt worden sind. Hierfür stellt der *AuthorityManager* die Methode *readAuthoritiesFromFile* zur Verfügung. Dieser delegiert diese Aufgabe an den *AuthorityReader*, der hierfür eine gleichnamige Methode bereitstellt. Dies ist konsistent mit der Forderung, dass der *AuthorityManager* die Zugriffsschnittstelle für die Autorisierungsarchitektur bildet. Folgende Veränderung in der Iterationsschleife liest somit die deklarierten Berechtigungen ein:

```
for (int i=roots.length-1; i>=0; i--) { // start at broker
    AuthorityManager.getAuthorityManager().readAuthoritiesFromFile
    (this.services, roots [i] + File.separator + "authorities.txt");
    dispatcher.init (roots[i] + File.separator + "handler.txt");
}
```

In diesem Zusammenhang ist bemerkenswert, dass der Autorisierungsarchitektur das *Services*-Objekt übergeben wird. Dieses Objekt bildet die zentrale Schnittstelle innerhalb der Broker-Architektur, um auf die Persistenzschicht (*Assets*) zugreifen zu können. Die Übergabe der *Services* dient der Initialisierung der elementaren Berechtigungen, wann immer diese benötigt werden.

Die wichtigsten Aufgaben des *AuthorityReaders* sind das Instanzieren der deklarierten Berechtigungen und das Zwischenspeichern der Zuordnungen von Prädikaten (String-Repräsentationen) zu Berechtigungen, und zwar über alle Projektebenen hinweg. Diese Zuordnung wird später verwendet, um die eigentliche Zuordnung von Handlern zu Berechtigungen durchzuführen. Darüber hinaus loggt die Klasse die Vorgänge während der Instanziierung, um ggf. Fehler schnell auffinden zu können.

Zu Beginn des Initialisierungsvorgangs wird die Methode ***readAuthoritiesFromFile*** des *AuthorityReaders* aufgerufen. Die *authorities.txt* der jeweiligen Projektebene wird zunächst als *Properties*-Tabelle eingelesen – ähnlich dem Vorgang bei der *handler.txt*. Der *AuthorityReader* hält eine weitere Tabelle, die als *Hashtable* implementiert ist und die dem Attribut ***authorityNamesAndInstances*** zugewiesen ist. Diese letzte Tabelle akkumuliert Prädikate (*Key der Hashtable*) und zugeordnete Berechtigungen (*Value der Hashtable*) über mehrere Iterationen. Für jeden Eintrag

der *Properties*-Tabelle wird zunächst geprüft, ob sich bereits ein gleichnamiges Prädikat in der *Hashtable* befindet, was unzulässig ist. Eine solche Verletzung wird als Fehler geloggt und ruft den Abbruch der jeweiligen Iteration hervor. Ist das Prädikat nicht in der *Hashtable* enthalten, so kann ein neues *Key-Value-Pair* in die *Hashtable* eingetragen werden, wobei der *Key* das Prädikat selbst und der *Value* eine zu generierende Berechtigung ist. Ob die Berechtigung zusammengesetzt oder elementar ist, hängt von der *Property* des Eintrags ab. Die Unterscheidung wird aufgrund von enthaltenen Schlüsselwörtern getroffen. Enthält die *Property* die Schlüsselwörter "NOT", "AND" oder "OR", so wird geschlussfolgert, dass es sich um einen zusammengesetzten Berechtigungsausdruck handelt. In diesem Fall wird die Methode ***createComposedAuthority*** aufgerufen, um den Ausdruck zu parsen. Für einen elementaren Berechtigungsausdruck wird die Methode ***createSimpleAuthority*** aufgerufen.

Die Methode *createSimpleAuthority* instanziiert den entsprechenden Berechtigungstyp, initialisiert die Berechtigung und trägt sie unmittelbar in die *authorityNamesAndInstances-Hashtable* ein. Der Vorgang und evtl. auftretende Fehler werden hierbei geloggt.

Parsen zusammengesetzter Berechtigungen

Handelt es sich um einen zusammengesetzten Berechtigungsausdruck, so wird die Methode *createComposedAuthority* aufgerufen. Diese Methode generiert zunächst ein Objekt der Klasse ***AuthorityTree***. Hierbei handelt es sich um ein Datenobjekt, das zwei Attribute enthält. Das erste Attribut wird von dem Prädikat selbst gebildet. Das zweite Attribut enthält den Wurzelknoten eines binären Baumes, welches aus dem Berechtigungsausdruck generiert wird. Objekte der Klasse *AuthorityTree* sind Zwischenrepräsentationen zusammengesetzter Berechtigungen. Diese Objekte werden in einer eigens implementierten *Collection* namens ***AuthorityTreeCollection*** gespeichert. Sie besitzt zudem einen eigens implementierten *Comparator* namens ***AuthorityTreeComparator***, der ein sehr spezielles Sortierkriterium für die *Collection* bereitstellt. Zusammenfassend lässt sich sagen, dass die zusammengesetzten Berechtigungsausdrücke der *authorities.txt* einer Projektebene jeweils in die Zwischenrepräsentation *AuthorityTree* überführt, in der *AuthorityTreeCollection* gespeichert und mit Hilfe des *AuthorityTreeComparator* sortiert werden.

Ein *AuthorityTree*-Objekt enthält den Wurzelknoten eines binären Baumes, der den Berechtigungsausdruck repräsentiert. Knoten dieses binären Baumes werden durch Objekte der Klasse ***AuthorityBinaryTreeNode*** verkörpert. Hier werden die Berechtigungsausdrücke geparkt und die innere Logik des Ausdrucks durch die Struktur des Baums beschrieben. Endknoten eines solchen Baums sind stets die Prädikate, die in dem Berechtigungsausdruck verwendet werden. Die Knoten, die nicht Endknoten sind, enthalten eine Typisierung entsprechend der logischen Operation, die sie verkörpern (NOT, AND, OR). Zusammenfassend lässt sich sagen, dass ein *AuthorityBinaryTreeNode*-Wurzelknoten einen Berechtigungsausdruck der *authorities.txt* repräsentiert. Jeder Wurzelknoten wird jeweils in einem *AuthorityTree*-Objekt gespeichert, das zusätzlich zu dem Berechtigungsausdruck auch das zugeordnete Prädikat repräsentiert.

Die Gründe für dieses komplexe Vorgehen bei der Instanziierung von zusammengesetzten Berechtigungen sind: Zunächst einmal wird die *Properties*-Tabelle, die die jeweilige *authorities.txt* einer Projektebene verkörpert, nicht notwendig in der Reihenfolge abgearbeitet, wie dies aufgrund der Reihenfolge der Prädikate anzunehmen wäre. Außerdem kann man nicht davon ausgehen, dass die

Prädikate in der *authorities.txt* stets richtig sortiert sind. Dies hat zur Folge, dass zum Zeitpunkt, in dem die Logik eines zusammengesetzten Berechtigungsausdrucks analysiert wird, nicht notwendig alle enthaltenen Prädikate zu Berechtigungen instanziiert worden sind, so dass hierfür eine Zwischenspeicherung und eine Abarbeitungsreihenfolge erforderlich ist⁵¹. Dies wird auf die bereits beschriebene Weise erreicht.

Im Ergebnis hat man am Ende der Methode *readAuthoritiesFromFile* alle elementaren Prädikate zu Berechtigungen instanziiert und für jeden zusammengesetzten Berechtigungsausdruck ein *AuthorityTree*-Objekt instanziiert. Am Ende dieser Methode wird somit eine weitere Methode aufgerufen, die das Instanziiieren der zusammengesetzten Berechtigungen durchführt: ***createComposedAuthorities***⁵². Die Methode läuft in einer Endlosschleife, solange die *AuthoritiesTreeCollection* nicht leer ist, d.h. solange nicht alle zusammengesetzten Berechtigungen instanziiert sind. Innerhalb der Iteration findet zunächst ein Sortiervorgang statt, durch den sichergestellt wird, dass sich am Anfang der *Collection* stets ein Ausdruck befindet, dessen verwendeten Prädikate bereits alle zu Berechtigungen instanziiert worden sind. Hierdurch kann stets das erste Element (ein *AuthorityTree*) aus der *Collection* genommen (und aus dieser gelöscht) werden, und eine zusammengesetzte Berechtigung durch einfaches Abarbeiten der Struktur des binären Baumes generiert werden. Da das Sortierkriterium sicherstellt, dass zu jedem vorkommenden Prädikat im binären Baum bereits eine Berechtigung existiert, können die entsprechenden Berechtigungen in die top-down zu generierende zusammengesetzte Berechtigung eingetragen werden. Diese Aufgabe wird von der Methode ***createComposedAuthority (AuthorityTree tree)*** innerhalb der Klasse *Authority* durchgeführt. Nachdem die zusammengesetzte Berechtigung instanziiert ist, wird sie wie auch die elementaren Berechtigungen in die *authorityNamesAndInstances-Hashtable* eingetragen. Da bei jeder Iteration ein Element aus der *Collection* entfernt wird, wird die Schleife nach dem Instanziiieren der letzten zusammengesetzten Berechtigung verlassen. Kommt es während der Instanziiierung der zusammengesetzten Berechtigungen zu internen Fehlern, so werden diese geloggt.

Nach dem Durchlaufen der Methode *readAuthoritiesFromFile* sind auf die beschriebene Weise alle Prädikate der *authorities.txt* einer Projektebene in entsprechende Berechtigungen überführt. Die *Hashtable authorityNamesAndInstances* akkumuliert dann sämtliche Prädikate und ihre zugeordneten Berechtigungen über mehrere Projektebenen hinweg, so dass am Ende der Iteration über die Projektebenen die *Hashtable* alle verwendbaren Prädikate enthält. Damit ist die Umwandlung der deklarativen Prädikate der *authorities.txt* in operative Berechtigungen hinlänglich erläutert.

Die Berechtigungszuordnung

Nachdem die Instanziiierung der deklarierten Berechtigungen erläutert ist, soll die Umwandlung der deklarativen Zuordnung innerhalb der Dateien *handler.txt* in die operative Zuordnung innerhalb des *HandlerAuthorityMappers* erläutert werden. Wie bereits dargestellt, erfordert diese Umwandlung eine Anpassung der Methode *init (String handlerTable)* innerhalb des *Dispatchers*.

⁵¹ Ein zweistufiger Prozess ist hierbei stets erforderlich.

⁵² nicht verwechseln mit *createComposedAuthority*

Die Methode *init* liest eine *handler.txt* als *Properties*-Tabelle ein. Da aufgrund der Autorisierungsarchitektur an den Klassennamen des Handlers das gewünschte Prädikat angefügt wird, wird die bisherige Initialisierung solche Einträge nicht bearbeiten können. Um den bisherigen Instanziierungscode für Handler weiterhin verwenden zu können, müssen die Einträge der Prädikate innerhalb der *Properties* entfernt werden. Gleichzeitig ist es allerdings erforderlich, diese Zuordnung für die Autorisierungsarchitektur zwischenspeichern. Diese Aufgabe wird von der Methode ***extractAuthorityNames (Properties properties)*** der Klasse *AuthorityManager* durchgeführt. Innerhalb der *init*-Methode wird diese Methode **vor** dem Instanzieren der Handler aufgerufen. Sie stellt das ursprüngliche Format innerhalb der *Properties*-Tabelle her. Die Zuordnung von Handlern zu Prädikaten wird in einer transienten Tabelle zwischengespeichert, die innerhalb des *AuthorityManagers* unter dem Attribut ***transientAuthorityMapper*** gehalten wird.

Nach dem Instanzieren der Handler substituiert die Methode ***putHandlersInsteadOfHandlerNames (Hashtable transientTable)*** innerhalb des *Dispatchers* die zuvor ausgelesenen Klassennamen innerhalb der transienten Tabelle durch die nun vorhandenen Handlerinstanzen. Die transiente Tabelle enthält somit nach dem Aufruf dieser Methode die Zuordnung von Handlerinstanzen zu Prädikaten von der *handler.txt* einer Projektebene. Dieser Vorgang wird im *Dispatcher* geloggt.

Zum Abschluss der Iteration in einer Projektebene werden die in der transienten Tabelle zwischengespeicherten Zuordnungen dem *HandlerAuthorityMapper* übergeben. Hierzu wird am Ende der *init*-Methode die Methode ***initTransientAuthorities*** der Klasse *AuthorityManager* aufgerufen. Innerhalb dieser Methode wird die transiente Tabelle an den *HandlerAuthorityMapper* übergeben und diese anschließend gelöscht.

Der folgende Programmcode zeigt die notwendigen Anpassungen innerhalb der *init*-Methode im *Dispatcher*:

```
// Einlesen der handler.txt einer Projektebene,
// diese wird in der Properties-Tabelle properties gehalten

// ursprüngliches Format wiederherstellen
AuthorityManager.getAuthorityManager().extractAuthorityNames
(properties);

// Instanziierung der Handler (verwendet properties)

// substituiert Handler-Klassennamen durch Handler-Instanzen
this.putHandlersInsteadOfHandlerNames
(AuthorityManager.getAuthorityManager().getTransientAuthorityMapper());

// transiente Tabelle in permanente Zuordnung überführen und löschen
AuthorityManager.getAuthorityManager().initTransientAuthorities();
```

Die eigentliche, permanente Zuordnung findet im *HandlerAuthorityMapper* statt. Die übergebene transiente Tabelle wird ausgelesen, die hierin vorhandenen Handlerinstanzen werden in der Zuordnungstabelle abgelegt. Hierbei werden die in der transienten Tabelle vorhandenen Prädikate mit Hilfe des *AuthorityReaders* auf Berechtigungen abgebildet, so dass die Zuordnungstabelle schließlich eine Abbildung von Handlerinstanzen auf Berechtigungen enthält. Der Abbildungsvorgang wird innerhalb der Klasse *HandlerAuthorityMapper* geloggt. Kommt es während dieser Zuordnung zu internen Fehlern aufgrund der Tatsache, dass Prädikate nicht

deklariert worden sind oder dass Berechtigungszuordnungen redefiniert werden, so werden diese Fehler ebenfalls geloggt.

Nachdem über sämtliche Projektebenen iteriert worden ist, um Handler und Berechtigungen zu instanzieren (Klasse *Broker*), wird eine Abschlussmethode aufgerufen, die bewirkt, dass auf die Zuordnungstabelle nur noch lesend zugegriffen werden kann. Die Autorisierungsarchitektur wird auf diese Weise nach dem Initialisierungsvorgang vor unerwünschten Veränderungen geschützt.

4.4 Zusammenfassung

Innerhalb dieses Kapitels ist die Autorisierungsarchitektur erarbeitet worden. Hierbei sind zunächst die Handler, die die implementierten Prozesse der Portalplattform verkörpern, als schützenswerte Ressourcen identifiziert worden. Im Anschluss ist auf die Repräsentation der Zugriffsrechte durch deklarative Prädikate und auf ihre programmatische Umsetzung durch Berechtigungen eingegangen worden. Die deklarative Kombinierbarkeit von Prädikaten und die programmatische Erweiterbarkeit durch neue Berechtigungstypen ist dabei ausführlich beschrieben worden. Auch auf die Abläufe während eines Prüfvorgangs ist eingegangen worden. Abschließend ist die technische Umsetzung der Initialisierung von Berechtigungen erklärt worden.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Diese Arbeit zeigt die Integration verschiedener Sicherheitskonzepte in die Portalplattform infoAsset Broker. Es sind konkret zwei Komponenten implementiert worden, nämlich die SWS- und die Autorisierungsarchitektur, die insgesamt vier verschiedene Sicherheitskonzepte umsetzen: Vertraulichkeit und Integrität der übertragenen Daten, Authentisierung der Kommunikationspartner und Autorisierung beim Aufruf implementierter Prozesse. Weitere Sicherheitskonzepte, die Beachtung verdienen, sind innerhalb dieser Arbeit ebenfalls identifiziert worden. Zum Teil werden hierfür in 5.2 Lösungsansätze vorgeschlagen. Dies gilt insbesondere für das Sicherheitskonzept Verbindlichkeit (vgl. 5.2.1). Darüber hinaus sind innerhalb dieser Arbeit eine Fülle von sicherheitsrelevanten Grundlagen zusammengetragen worden, die eine Sensibilisierung für das Thema Sicherheit fördern.

Im Ergebnis erhält man eine Portalplattform, die gegenüber der Ausgangssituation erheblich sicherer ist und die durch die in dieser Arbeit gemachten Vorschläge noch sicherer gemacht werden kann.

5.2 Ausblick

Die im Rahmen dieser Arbeit implementierten Sicherheitsmaßnahmen bilden beim infoAsset Broker den Grundstein für die kontinuierliche Weiterentwicklung der Sicherheitsarchitektur. Eine solche Weiterentwicklung kann einerseits durch Erweiterung der Sicherheitsarchitektur, d.h. durch das Erkennen neuer Risiken und durch die Implementierung neuer Sicherheitskonzepte, stattfinden und andererseits durch die Verbesserung bereits implementierter Maßnahmen. In diesem Kapitel sollen Beispiele für sinnvolle Weiterentwicklungen dieser Architektur genannt werden. Dies verdeutlicht auch, wieso die Integration von Sicherheitsmaßnahmen als Prozess zu verstehen ist.

5.2.1 Das Sicherheitskonzept Verbindlichkeit

Die integrierte Umsetzung des Sicherheitskonzeptes Verbindlichkeit stellt mit Sicherheit eine ganz besondere Herausforderung dar. Dies gilt insbesondere dann, wenn rechtsverbindliche Verträge bzw. zeichnungspflichtige Geschäftsprozesse in einer Weise umgesetzt werden sollen, dass sie juristisch, d.h. vor einem ordentlichen Gericht, Bestand haben. Eine Umsetzung dieses Konzeptes würde somit nicht nur technische Fragestellungen berühren, sondern auch die Gesetzgebung des jeweiligen Staates, in dem die Portalplattform zum Einsatz kommen soll. Darüber hinaus kennt der Informationsfluss im Internet keine staatlichen Grenzen, so dass bei einem internationalen Einsatz ebenfalls die diesbzgl. juristischen Fragen zu klären wären.

Aus technischer Sicht bilden digitale Zertifikate eine wichtige Grundlage für die Umsetzung der digitalen Signatur – also der Maßnahme, die das Sicherheitskonzept Verbindlichkeit umzusetzen vermag. Digitale Zertifikate sind innerhalb dieser Arbeit nicht zuletzt aus diesem Grund verwendet und so umfangreich behandelt worden.

Geht man stärker in technische Details, so ist zu konstatieren, dass eine digitale Signatur von einem Zertifikatsinhaber geleistet wird, indem dieser einen Hashwert⁵³ mit seinem privaten Schlüssel asymmetrisch chiffriert. Möchte man das Konzept Verbindlichkeit in integrierter Weise umsetzen, so sind prinzipiell zwei Wege möglich: die clientseitige und die serverseitige Signatur. Bei der clientseitigen Signatur wird der Hashwert clientseitig signiert und zum Server übertragen. Bei der serverseitigen Signatur wird ein serverseitiger Prozess durchgeführt, durch den der Hashwert beim Server direkt signiert wird.

Beide Möglichkeiten haben Vor- und Nachteile: Eine serverseitige Signatur ist verhältnismäßig leicht zu implementieren, da serverseitig ja die gesamte Java-Umgebung zur Verfügung steht. Denkbar wäre, einen dezidierten Signatur-Keystore mit eigens für die digitale Signatur geschaffenen Zertifikaten bereitzustellen. Bei den verwendeten Zertifikaten könnten selbstsignierte Zertifikate ausreichen, so dass diese mit Hilfe von keytool erstellt werden könnten und keine zusätzliche Kostenbelastung darstellen würden. Problematisch ist hier u.U. die Frage nach der juristischen Verbindlichkeit.

Eine clientseitige Signatur ist erheblich komplizierter. Hier muss berücksichtigt werden, dass clientseitig Browser vorliegen, deren programmatische Möglichkeiten stark eingeschränkt sind. Hier gilt es also zunächst zu analysieren, wie ein Hashwert unter Berücksichtigung der jeweiligen Schnittstelle (z.B. PKCS#11) von dem im Browser verwendeten Sicherheitsgerät signiert werden kann. Da in diesem Fall die Zertifikate für die clientseitige Authentisierung bereits vorliegen, existiert hier kein zusätzlicher administrativer Aufwand, wie es bei der serverseitigen Signatur der Fall wäre. Darüber hinaus erfüllt dieser Weg wohl eher die juristischen Bedingungen als der zuvor beschriebene.

Eine weitere Möglichkeit, das Sicherheitskonzept Verbindlichkeit umzusetzen, wäre eine nicht integrierte Lösung. Bei diesem Lösungsansatz käme Fremdsoftware zum Einsatz, um digitale Dokumente zu signieren bzw. um geleistete Signaturen zu verifizieren. Der infoAsset Broker würde in diesem Zusammenhang lediglich die Transportplattform für derartige Dokumente bilden. Zeichnungspflichtige Geschäftsprozesse wären der Portalplattform zunächst auf Dokumente abzubilden, damit diese wiederum mit der Fremdsoftware signiert werden könnten. Als Beispiel für den Einsatz von Fremdsoftware sind die Produkte der Fa. Mobile Technologies International Ltd. zu nennen, die über ihre Website www.pdfree.de erreichbar ist. Dieses Unternehmen bietet Signatursoftware an, die eigenen Angaben zufolge die juristischen Bedingungen innerhalb Europas erfüllt. Bei den signierten Dokumenten handelt es sich dabei um pdf-Files. Darüber hinaus ist die Software, die zur Verifikation der Signatur dient, zeitlich unbegrenzt kostenlos nutzbar.

5.2.2 Authentisierung: Verbesserungsvorschläge

Durch die in Kap. 3 erarbeitete *SWS-Architektur* werden verschiedene Authentisierungskonzepte umgesetzt. Wie dort erläutert, ist ein entscheidendes Entwurfskriterium, die Erweiterbarkeit der Architektur zu ermöglichen. Eine denkbare Verbesserung betrifft somit die Einführung neuer Authentisierungskonzepte. Die bereits implementierten SSL-Server können hierbei als Referenzimplementierungen verwendet werden, um ggf. neue clientseitige Sicherheitsgeräte in gebührender Weise zu berücksichtigen.

⁵³ dem Hashwert des Vertragswerkes

Ein wichtiger Punkt bei der zertifikatsbasierten Client-Authentisierung ist die serverseitig sichere Unterscheidung der clientseitig verwendeten Sicherheitsgeräte. Da der Server tatsächlich nur das vom Client vorgelegte Zertifikat sieht, muss er aufgrund der dort enthaltenen Information über die Gewährung oder die Ablehnung eines Verbindungsaufbaus entscheiden. Hierfür sind im Einzelfall klare Anwendungsrichtlinien festzulegen. Bisher trifft ein Server diese Entscheidung allein aufgrund des jeweils verwendeten Truststores. Dies impliziert somit, dass die Vertrauensanker für einen Server, der den Softwaresicherheitsdienst nutzt, anders als die eines Servers, der eine Authentisierung durch die Verwendung einer SmartCard vorsieht, sein müssen. Dies muss nicht notwendigerweise so sein: Allerdings findet eine sinnvolle Auswahl unter den Möglichkeiten nicht allein durch die Betrachtung von technischen, sondern auch von betriebswirtschaftlichen Gesichtspunkten statt. Die zu verwendenden Client-Zertifikate müssen entweder von einer anerkannten CA stammen oder werden von einem eigenen Certificate Server⁵⁴ produziert. Beide Wege sind mit nicht zu vernachlässigenden Kosten verbunden und müssen einer Abwägung unterzogen werden. Es wird an dieser Stelle noch mal darauf hingewiesen, dass die in dieser Arbeit zu Testzwecken verwendeten Zertifikate alle kostenlos gewesen sind, da die zu ihrer Ausstellung gehörigen Richtlinien praktisch keine echte Identitätsverifikation vorsehen. Verwendet man die zertifikatsbasierte Client-Authentisierung des infoAsset Brokers für eine betriebswirtschaftliche Anwendung oder gar im eGovernment-Bereich, so müssen natürlich auch die Ausstellungsrichtlinien der verwendeten Zertifikate den Bedürfnissen der jeweiligen Anwendung entsprechen.

Aus rein technischer Sicht ist eine weitere Verbesserung der SWS-Architektur denkbar: Möchte man weitere Konzepte während des SSL-Handshakes ermöglichen, so kann eine Instanziierung der Klasse *SSLContext*, die von der Standardinstanziierung abweicht, durchgeführt werden. Dies ist beispielsweise sinnvoll, wenn man eine eigene Implementierung eines TrustManagers nutzen möchte. Ein mögliches Anwendungsszenario für eine solche Maßnahme ist gegeben, wenn man Certificate Revocation Lists (CRLs) implementieren möchte, um gesperrte Zertifikate erkennen zu können.

5.2.3 Autorisierung: Verbesserungsvorschläge

Die Erweiterungsmöglichkeiten innerhalb der Autorisierungsarchitektur sind in Kap. 4 dargestellt worden. Die Architektur sieht vor, daß weitere Berechtigungstypen programmatisch eingefügt werden. Die bereits implementierten Berechtigungstypen können hierbei als Referenzimplementierungen dienen. Dies kann letztlich soweit gehen, daß zu jedem Handler die spezifische Berechtigung allein programmatisch definiert wird.

Eine Weiterentwicklung, die wesentlich umfangreicher als die Implementierung einzelner neuer Berechtigungstypen ist, wäre die Implementierung weiterer Kombinationsmöglichkeiten von Prädikaten. Relativ einfach wäre hierbei noch die Implementierung weiterer logischer Operatoren (z.B. XOR). Eine weitere Schwierigkeitsstufe erreicht man, wenn man nicht-boolsche Operationen ermöglichen möchte (z.B. Mengenoperationen). Diese zuletzt genannte Erweiterung würde nicht nur eine umfangreiche Veränderung des Programmcodes erfordern, sondern möglicherweise auch eine neue Konzeption der Architektur.

⁵⁴ eigene PKI

Weitere Verbesserungen aus Autorisierungssicht ergeben sich aus einer veränderten konzeptionellen Betrachtung: In der erarbeiteten Autorisierungsarchitektur besteht die schützenswerte Ressource aus den Handlern, die die implementierten Abläufe verkörpern. In gleicher Weise könnte der Ansatz unternommen werden, den Zugriff auf die eigentlichen Informationsobjekte beim infoAsset Broker – die Assets – zu regulieren. Die hier erarbeitete Autorisierungsarchitektur kann der Umsetzung dieses weitergehenden Konzepts als Grundlage dienen.

Um die Administration von vorhandenen Berechtigungen zu erleichtern, wäre es darüber hinaus denkbar, eine höhere Integration der Berechtigungen in die Architektur der Portalplattform zu erwirken. Dies könnte beispielsweise durch die Umwandlung von Berechtigungen zu Assets erreicht werden, d.h. die Berechtigung als spezialisiertes Geschäftsprozessobjekt zu verstehen. Hierbei ist allerdings zunächst zu untersuchen, ob durch diese Strategie nicht neue Sicherheitsrisiken entstehen können.

Literatur- und Quellenverzeichnis

- [Amor01] Die E-Business-(R)Evolution, Daniel Amor, Galileo Press GmbH, 2001
- [Arm02] Sicherheit und Benutzerverwaltung im Unternehmensportal (PowerPoint Präsentation), Martina Armbruster, SAP Portals, SAP AG, 2002
- [Atm01] Atmel Technical SmartCard White Papers, Atmel Corporation, 2001
- [BEASys01] BEA WebLogic Portal – Security Guide Version 4.0 (eBook, e-docs.bea.com), BEA Systems, Inc., 2001
- [BEASys02] BEA WebLogic Server – Introduction to WebLogic Security, BEA Systems, Inc., 2002
- [BSI01] Theoretische Grundlagen von Sicherheitsstrukturen, BMWi – BMI – BSI (www.sicherheit-im-internet.de), 2001
- [DPSign] Deutsche Post Signtrust GmbH (www.signtrust.de), Web Site Root CA & SmartCard Provider
- [Eck02] Thinking In Java 3rd Edition (HTML-Ausgabe, www.BruceEckel.com), Bruce Eckel, MindView, Inc., 2002
- [Ess02] CrypTool (PowerPoint Präsentation, www.cryptool.de), Bernhard Esslinger et al., Deutsche Bank, 2002
- [FNMT] Fábrica Nacional de Moneda y Timbre – Real Casa de la Moneda (www.cert.fnmt.es), Web Site für e-Government in Spanien (Offizielle Root CA)
- [Fow98] UML konzentriert (Deutsche Übersetzung), Martin Fowler & Kendall Scott, Addison Wesley Longman Verlag GmbH, 1998
- [Global] GlobalSign NV / SA (www.globalsign.com), Web Site Root CA
- [Götz99] SSL – Secure Socket Layer (Vorlesungszusammenfassung, www.tfh-berlin.de/~toby/vs/ssl), Tobias Götz, TFH-Berlin, 1999
- [Hör02] Portal Verbund Whitepaper, Rainer Hörbe, BMI, 2002
- [Holm01] Sicherheitstechnologien und –konzepte für Enterprise Portals, Michael Holman, SAP Portals (www.sap.de/enterpriseportals), SAP AG, 2001
- [iAB00] infoAsset Broker Kurzschulung, infoAsset AG (www.infoasset.de), 2000
- [iAB01a] The infoAsset Broker, Technical White Paper, infoAsset AG (www.infoasset.de), 2001
- [iAB01b] infoAsset Broker Documentation, Together-generated documentation, 2001
- [iAB02] Funktionen infoAsset Broker (Excel-Dokument), infoAsset AG, 2002
- [iABonl] Der infoAsset Broker: Software für Wissensmanagement und Content Commerce (Online-Information), infoAsset AG (www.infoasset.de → Produkte)
- [J2Sun02] Java 2 SDK Standard Edition Version 1.4.1. (Online Dokumentation, java.sun.com), Sun Microsystems, Inc., 2002
- [JavaScr01] JavaScript – Programmieranleitung, Web Site (js-tut.aardon.de)

- [JavaScr02] JavaScript Online Hilfe (Automatische Weiterleitung), Web Site (www.sachen-fuer-webmaster.de/printfeature.php?artid=64)
- [JavaScr03] JavaScript Online Hilfe (Automatische Weiterleitung), Web Site (www.suchmaschinentricks.de/technik/weiterleitungen.html)
- [Kov02] IBM WebSphere V4.0 Advanced Edition Security (1st Edition, eBook, www.ibm.com/redbooks), Peter Kovari et al., IBM Corporation, 2002
- [Krüg02] Handbuch der Java-Programmierung 3. Auflage (HTML-Ausgabe, www.javabuch.de), Guido Krüger, 1998 – 2002
- [Mat00] E-Commerce: Concepts and Technologies (Vorlesungsunterlagen), Florian Matthes, TUHH AB Softwaresysteme (www.sts.tu-harburg.de), 2000
- [net01] netlife banking server & netlife HBCI Online Client (Product White Papers), netlife AG (www.netlife.de), 2001
- [Nets02] Netscape 7.0 Online Hilfe, Netscape Communications Ireland, Inc. (www.netscape.de), 2000 – 2002
- [Oaks01] Java Security 2nd Ed., Scott Oaks, O'Reilly & Associates, Inc., 2001
- [Pau00] mySAP.com – Sicherheit im e-Business (PowerPoint Präsentation), Sachar Paulus, SAP AG, 2000
- [Pist99] Java 2 Network Security (eBook, www.redbooks.ibm.com), Marco Pistoia et al., IBM Corporation, 1999
- [Rei02] Chipkarten in der IT-Sicherheit (Seminarvortrag), Philipp Reichmuth, AB Informatik 3, 2002
- [RSA01] PKCS#11 v2.11: Cryptographic Token Interface Standard, RSA Security, Inc. (www.rsasecurity.com), 2001
- [SAP01] mySAP Enterprise Portals (Funktionen, Rollen, Sicherheit), (www.sap.com → Deutschland → Lösungen → mySAP Enterprise Portals), SAP AG, 2001
- [SAP02a] mySAP Technology, Voraussetzung für eine offene E-Business-Integration – Ein Überblick (Version 1.0), SAP White Paper, SAP AG, 2002
- [SAP02b] mySAP Technology, Portal-Infrastruktur: Benutzerorientierte Integration und Zusammenarbeit (Version 1.1), SAP White Paper, SAP AG, 2002
- [SAP02c] mySAP Technology, Sicherheit: Sichere Geschäftsabwicklung in offenen Umgebungen (Version 1.1), SAP White Paper, SAP AG, 2002
- [SAPMag01] sapinfo.net/_ NR.82 / MAI 2001 (Das SAP-Magazin), SAP AG, 2001
- [SAPPub01] Digitale Signatur und Smartcard im E-Business, mySAP Public Sector, SAP AG, 2001
- [SAPSec01] Digitale Signaturen (PowerPoint Präsentation), SAP Security Competence Team, SAP AG, 2001
- [Ste02] SmartCards – Security and Integration, Günter Stecker, LOGICO Smartcard Solutions (www.smartcard-solutions.com), 2002
- [TCTrust] TC Trustcenter AG (www.trustcenter.de), Web Site Root CA

- [Uti02a] SafeGuard Biometrics Version 1.60 (Benutzerhandbuch des SmartCard-Systems), Utimaco Safeware AG (www.utimaco.de), 2002
- [Uti02b] Utimaco Whitepapers, Utimaco Safeware AG (www.utimaco.de), 2002
- [Veit00] Sicherheit im Internet (Handout zum Vortrag auf der „Systems“), Thomas Veit, BSI, 2000
- [VeriSign98] Implementing Web Site Client Authentication Using Digital Ids and the Netscape Enterprise Server 2.0, VeriSign, Inc. (www.verisign.com/repository/clientauth/ent_ig.htm), 1998
- [Wong00] HTTP kurz und gut (Deutsche Ausgabe), Clinton Wong, O'Reilly Verlag, 2000
- [Zim97] Kryptologie – Sicherheit in Datennetzen (Vorlesungsskript), Karl-Heinz Zimmermann, TUHH AB Technische Informatik VI, 1997